

PYTHON - DATA SCIENCES

NUMPY - MATPLOTLIB

SOMMAIRE

Sommaire	1
1- Introduction aux data-sciences	6
Introduction.....	6
Triple compétences Data-Sciences.....	6
L'écosystème data-sciences python	8
Les environnements technologiques data-sciences.....	9
Sources	10
Datascientist.com	10
MOOC.....	10
OCR	10
W3School	10
Liste d'outils.....	11
Notebook	12
Webographie	12
Présentation.....	12
Aperçus – mise en bouche (10-19)	16
Aperçus de numpy et pandas	16
numpy	17
Notebook	17
Installation et utilisation d'un notebook	17
Vitesse de calcul.....	18
pandas	19
Notebook	19
Installation	19
Affichage d'un tableau CSV : équivalent d'un SELECT *	19
Forme ou format d'un tableau : nombre de lignes et de colonnes.....	19
Sélection de colonnes : équivalent d'un SELECT pclass, survived, sex, age	20
GROUP BY : taux de survie par sexe et par classe	20
GROUP BY : statistiques diverses.....	21
GROUP BY : statistiques diverses et chatgpt	22
PIVOT.....	23
Autres usages :	24
Variante seaborn du PIVOT.....	25
Matplotlib - visualisation des résultats	26
Notebook	26
Notebook	26
2 - Numpy	27
Présentation (5p - 27-31).....	27
Références officielles.....	27
Webographie	27
Installation pip ou pip3	28
Utilisation de numpy	28
Rappels introductifs.....	29
Origine.....	29
Philosophie.....	29

Principes.....	29
Bilan	29
4 notions fondamentales numpy	30
Programmation fonctionnelle	31
Le type ndarray (25p - 32-56).....	32
Qu'est-ce que c'est ?	32
Principes.....	32
Schémas	33
Documentation officielle	34
Méthodes de manipulation	34
Premières fonctions de création d'un tableau numpy	35
Fonction array()	35
Forme, dimension et taille d'un tableau : shape, ndim, size.....	35
Fonction ones(), zeros(), full()	36
Fonction random.randint()	37
Fonction random.random()	37
Fonction arange().....	38
Fonction eye()	38
Fonction empty()	39
Fonction copy()	39
Fonction reshape().....	40
Exercices	41
Matrice	41
Cube	41
Première approche de la programmation vectorielle	42
Variantes algorithmiques.....	42
Types, ndarray et dtype.....	43
rappels sur les types python	43
types numpy	43
ndarray.....	43
dtypes.....	44
Les types scalaires	45
liste des types scalaires :	45
Exemple : création d'un tableau numpy d'un type scalaire numérique	45
Transtypage automatique et limites.....	46
nan : not a number : valeur invalide.....	46
Principe : éviter les conversions implicites !.....	47
Changement de type - transtypage	47
Rappels de transtypage python	47
Les autres types scalaires	48
Présentation.....	48
Les autres types scalaires.....	48
Les types str et bytes	49
Les types composites.....	51
En résumé : synthèse des attributs et des fonctions	52
Attributs de base : size, shape, ndim, itemsize	52
Fonctions de création de base :	53
Fonctions de base : min, max, sum, mean, quantile, prod.....	54
Fonctions de base : tri.....	54
Exercices	55
Exercice	55
Exercice	55
Exercice	55
Exercice mean, max, reshape	56
Indexation avancée (57-75 : 19p).....	57
Présentation	57
Slicing et vue.....	58

Slicing multidimensionnel	58
Slicing et vue	58
Notion de vue	59
Inverse du slice : concatenate() et _r	59
Slicing de colonnes.....	60
Modifier un slicing de colonnes.....	61
Extraire un bloc par slicing.....	61
Modifier un slicing de bloc.....	61
Passer d'une ligne à une colonne par slicing.....	62
Exercices	63
Exercice	63
Exercice masque zeros reshape	63
Précisions sur le reshaping : changement de dimension	64
Principes.....	64
Méthode inverse : « aplatis » un tableau : ravel()	64
Variante : création et reshape enchainées :	65
Exercices	66
Exercice reshape de base.....	66
Exercice Matrice.....	66
Exercice Cube.....	66
Les tableaux booléens	67
Principes.....	67
Création d'un tableau de booléens avec une opération de vectorisation	67
Sum, any et all dans un tableau de booléens	68
Rappels : mean, max, min, sum, size	68
& et avec les tableaux de booléens combinés entre eux	69
Compléments sur all() et any()	70
Indexation avancée.....	71
Présentation.....	71
Exemples	71
Modification de certaines valeurs	72
Création d'un tableau par indexation.....	72
np.where() : selection	73
Exercices	75
Vectorisation (11p - 76-86)	76
Présentation	76
Principes algorithmiques	76
Exemple et constatation de l'optimisation	77
ufunc	78
Multiplication de matrices « traditionnelle ».....	79
Attention ! Ca n'a rien à voir !	79
Fonction dot() : multiplication de matrice.....	79
Utilisation d'un paramètre en sortie	80
Remarques sur le timeit	81
Modification de certains éléments : méthode at()	82
Statistiques par colonne ou par ligne : agrégation.....	83
Principes.....	83
Exemple.....	83
nan - nansum() – isnan() – invert() – logical_not()	84
Principes.....	84
Exemple 1.....	84
Compter les éléments nan ou les non nan d'un tableau.....	85
Écrire nos propres fonctions vectorisées	86
Principes.....	86
Décorateur @np.vectorize.....	86
Broadcasting (9p - 87-95)	87

Principes	87
Exemples.....	87
Broadcasting entre matrices	88
Principe général	88
Exemple 1 :	88
Exemple 2 :	89
Exemple 3 :	90
Application de broadcasting	91
Valeurs possibles pour 2 dés	91
Fonction unique() : nombre d'occurrences de chaque valeur dans un tableau	91
Valeurs possibles pour 3 dés	92
Exercices	93
Exercice	93
Exercice : Le damier	93
Exercice : la fonction <code>fij : tab[i, j] = 100*i + 10*j + offset</code>	94
Exercice : la pyramide	95
Fonctions et notions subtiles (10p - 96-105)	96
Fonction indice()	96
Fonction meshgrid()	97
Principe	97
Explication par un exemple :	97
Exemple plus subtil :	98
Fonction linspace()	99
Variante 1.....	99
Variante 2.....	100
Variante 3 : avec le décorateur <code>@np.vectorize</code>	101
Notion de masque	102
On va afficher une image d'un centre noir qui va en dégradé vers le gris.....	102
On va afficher des rayures noires en diagonale	103
Les diagonales vont servir de masque sur l'affichage de départ.....	104
Listes de fonctions plus ou moins subtiles :	105
3 – Matplotlib	106
Références.....	106
Matplotlib	106
MOOC	106
matplotlib - 2D (11p - 106-116)	107
Bases	107
Imports habituels	107
Changer la taille par défaut des figures matplotlib	107
plt.plot.....	108
Mise en page	111
Plusieurs subplots : <code>plt.subplot</code>	111
Mise en page : option 1 => à éviter	112
Mise en page : option 2 => à utiliser.....	113
Variantes d'affichage.....	114
plt.hist.....	114
plt.scatter	115
plt.boxplot.....	116
matplotlib 3D.....	117
Bases	117
Un premier exemple : une courbe.....	117
Axes3DSubplot.scatter.....	118
Axes3DSubplot.plot_wireframe	119
Axes3DSubplot.plot_surface.....	119
Axes3DSubplot.plot_trisurf	121
Axes3DSubplot.contour	124

Axes3DSubplot.contourf.....	125
Axes3DSubplot.add_collection3d.....	125
Axes3DSubplot.bar.....	126
Axes3DSubplot.quiver	127

Editions : juin 2022, décembre 2024

1- INTRODUCTION AUX DATA-SCIENCES

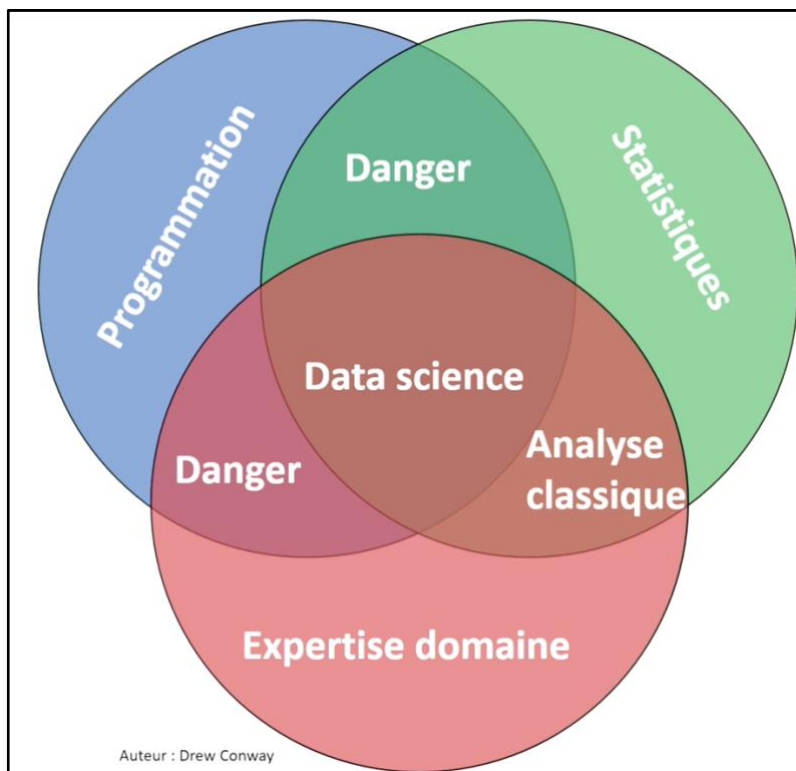
Introduction

Triple compétences Data-Sciences

- **Datasciences** : triple compétence :

- **statistiques** (= mathématiques),
- **programmation** (python),
- expertise domaine (**métier**)

⇒ Un data-scientiste maîtrise ces trois domaines. Mais on peut aussi intervenir uniquement dans certaines parties.



- **Le problème de la seule double compétence :**

- **stats + métier** = statistiques classiques. Ca marche mais il n'y a pas la puissance et la rapidité (de calcul et de présentation) de la programmation.
- **prog + métier** = **danger !** il manque les fondements statistiques et mathématiques !
- **prog + stat** = **danger !** il manque le métier d'application qui permet de cadrer les questions.

⇒ **Exemple en machine learning :**

- il y a bien sûr des stats et de la prog.
- Mais il aussi faut préparer ses données et choisir son algo d'apprentissage, ce qui passe par une compétence métier d'application.

- **Programmation :** Les briques de bases sont NumPy et Pandas.

- **Remarque :**

- ⇒ Un data-analyste se place plutôt du côté de la programmation et de l'expertise métier.
- Il lui manque un fondement mathématique-statistique pour l'analyse des données.
 - Toutefois, la puissance des outils de programmation permet déjà d'obtenir beaucoup de résultats.

L'écosystème data-sciences python

- L'écosystème data sciences python est en pleine effervescence.
 - **Numpy** : <https://numpy.org/> (Numerical Python)
 - **Pandas** : <https://pandas.pydata.org/>
 - **Scikit-learn** (machine learning) : <https://scikit-learn.org/stable/>
- Et toujours :
 - **Python** : <https://www.python.org/>
 - http://bliaudet.free.fr/article.php3?id_article=364
 - <http://bliaudet.free.fr/IMG/pdf/Python.pdf>
 - **Matplotlib** : <https://matplotlib.org/>
 - **Jupyter** : <https://jupyter.org/>
 - <https://jupyter.org/install>
 - Le Classique : pip install notebook
 - Jupyter notebook
 - pip show notebook
 - pip uninstall notebook
 - jupyter notebook
 - <http://localhost:8888/tree> : accès à votre disque dur !
 - pip install jupyterlab :
 - pip show jupyterlab
 - pip uninstall jupyterlab
 - jupyter lab
 - <http://localhost:8888/lab>
- A noter :
 - **SciPy** : <https://scipy.org/>
 - SciPy propose des algorithmes fondamentaux pour le calcul scientifique en Python.
 - Il intègre du NumPy

Au départ. : BI (Business Intelligence = informatique décisionnelle) : c'est du traitement de données à base de SQL.

⇒ SQL Server de Microsoft est orienté BI.

Ensuite : data science, big data, machine learning, data-mining.

⇒ Ecosystème Python

⇒ Scala (langage) et Spark (framework)

⇒ Hadoop

⇒ Talend

Sources

- Il y en a beaucoup sur internet ! C'est ce qui fait la force du Python !
- On en cite 4, de façon subjective.

Datascientist.com

- <https://datascientest.com/data-analysis-tout-savoir>
- La Data Science repose sur des calculs scientifiques d'une haute complexité.
- Pour effectuer ces calculs, les Data Scientists ont besoin d'outils puissants.
- Les bibliothèques Numpy, Pandas (SQL), Scikit-learn (machine learning) pour Python sont de précieuses ressources.

MOOC

- https://lms.fun-mooc.fr/courses/course-v1:UCA+107001+session02/courseware/82ac208ce93b479f90a5a65e04753f5a/c02cefb8de8c44fdbf48b862cf53fe87/1?activate_block_id=block-v1%3AUCA%2B107001%2Bsession02%2Btype%40vertical%2Bblock%40f059067f6e0f43c997c53c842d1ce067

OCR

- <https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/5560976-familiarisez-vous-avec-lecosysteme-python>

W3School

- <https://www.w3schools.com/python/>
- <https://www.w3schools.com/python/numpy/default.asp>
- etc.

Liste d'outils

- **Python** : 1994. Langage ancien et mature. Simple, flexible et généraliste. Pas adapté pour l'optimisation (la rapidité de certains calculs).
- **Numpy** : 2006. Librairie de référence pour **manipuler des tableaux en Python**. Tableaux multidimensionnels.
⇒ C'est très performant, particulièrement la vectorisation, au détriment d'une certaine simplicité.
- **Pandas** : 2008. Spécialisé et plus complexe. C'est la librairie de référence pour **ajouter des labels aux index des tableaux Numpy**, et ainsi les rendre plus explicites. De plus, on trouve dans Pandas les **opérations classiques de BD** : regroupement (group by), jointure, pivot.
⇒ C'est très performant, particulièrement la vectorisation, au détriment d'une certaine simplicité.

Numpy et Pandas, c'est ce qui se fait de mieux actuellement.
Ce n'est pas parfait. Il faut faire avec.

- **Notebook** : du texte formaté mélangé avec du code et avec lequel on peut interagir avec nos données.
⇒ C'est l'environnement préféré de la communauté datascience car ils permettent de faire des « runnable papers » : des papiers exécutables !

Webographie

➤ *MOOC*

<https://lms.fun-mooc.fr/courses/course-v1:UCA+107001+session02/courseware/b4a443f89e7145d79355eee81c358718/1125c3331b9443a08e594ec41c796e2b/#semaine-1-introduction-au-mooc-et-aux-outils-python-child>

➤ *OCR*

<https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/5560976-familiarisez-vous-avec-lecosysteme-python>

➤ *Guide d'utilisation*

<https://pyspc.readthedocs.io/fr/latest/03-guide/>

Présentation

➤ *Runnable papers : notebook*

- Les notebooks sont l'environnement préféré de la communauté datascience car ils permettent de faire des « runnable papers » : des papiers exécutables !
 - ⇒ Du texte formaté mélangé avec du code et avec lequel on peut interagir dynamiquement avec nos données.
 - ⇒ Le notebook, c'est le HTML du datascientist python.

➤ *Jupyter*

- Jupyter est un serveur qui permet de fabriquer nos notebooks dans un navigateur.
- Jupyter : Julia + Python + R

➤ *Installation pip ou pip3*

- pip3 install notebook.

➤ *Jupyter : <https://jupyter.org/>*

- <https://jupyter.org/install>
 - ⇒ Le Classique : pip install notebook
 1. Jupyter notebook
 2. pip show notebook
 3. pip uninstall notebook
 - 4. jupyter notebook
 - 5. <http://localhost:8888/tree> : accès à votre disque dur !
- ⇒ pip install jupyterlab :
 1. pip install jupyterlab
 2. pip show jupyterlab
 3. pip uninstall jupyterlab

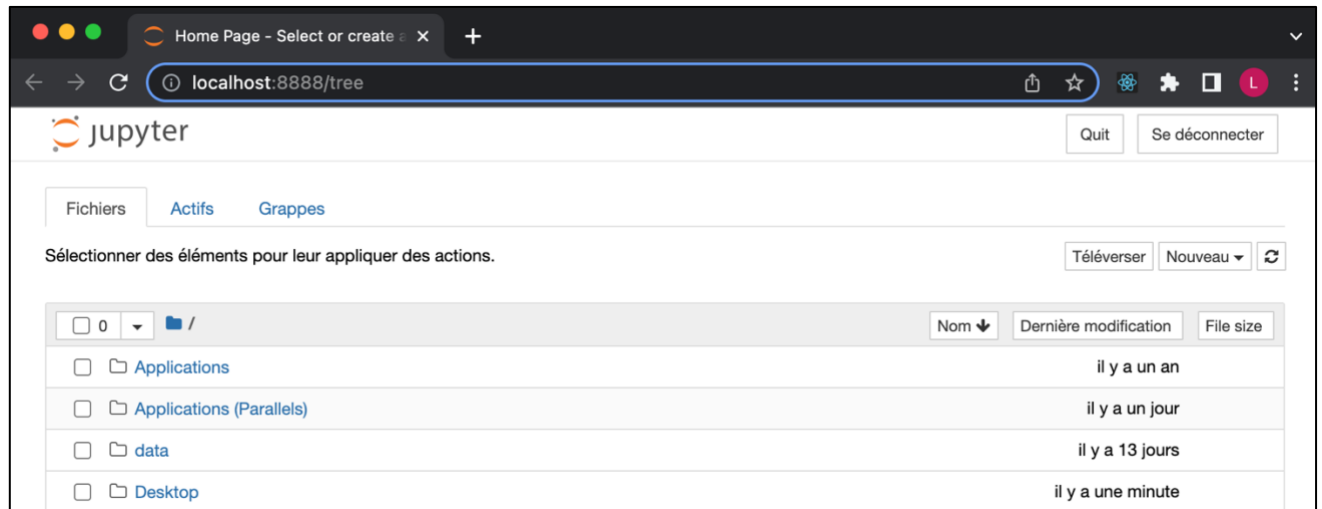
4. jupyter lab
5. <http://localhost:8888/lab>

➤ *Démarrage de jupyter – utilisation d'un notebook*

```
C:> jupyter notebook
```

⇒ Ca démarre un serveur local pour jupyter

⇒ Ca ouvre un client dans un navigateur : <http://localhost:8888/tree>



- On peut enregistrer les notebooks sur son disque dur

➤ Usages

- Nouveau :



- Notebook : mélange entre du texte et du code
 - ⇒ On met du texte : du bla bla explicatif, du code, on peut exécuter le code et le résultat s'affiche dans le notebook, à la suite du code.
 - ⇒ On peut effacer tous les affichages de résultats si on veut.

Ajouter du texte ou du code

- Le plus

Taper du code

- maj+entrée : fait descendre et exécute le code de la ligne
 - Pour effacer les exécutions : Noyau / redémarrer et effacer les sorties
 - Pour relancer toutes les exécutions : Noyau / redémarrer et tout exécuter
- On peut importer des fonctions écrites dans des fichiers .py : `from mon_fichier import ma_fonction`

Taper du texte

- On peut choisir le type de la cellule : du code python ou du texte (titre ou markdown pour du HTML)
 - On peut choisir un titre : # devant le texte, ##, ### pour différents niveaux de titre
 - On peut choisir un paragraphe

Supprimer du texte ou du code

- Le ciseau (couper)

Auto-complétion

- Auto-complétion : taper tab en cours de saisie.

Enregistrements .ipynb, .py et .html

- On peut enregistrer les notebooks sur son disque dur : .ipynb
- On peut enregistrer le notebook au format HTML (fichier / télécharger au format HTML) : très pratique. Ou python (pas très pratique mais ça peut être utile).

Aperçus de numpy et pandas

- On va montrer quelques exemples de Numpy et Pandas pour voir ce qu'on peut faire avec.
⇒ Le mieux sera de tester les exemples directement pendant le cours.
- **Numpy :**
 - ⇒ pour des tableaux python, multidimensionnels, avec des objets de même type.
 - L'exemple de base, c'est un tableau à 1 ou 2 dimensions avec des entiers, des réels ou des booléens.
 - ⇒ rapide grâce à la vectorisation.
- **Pandas :**
 - ⇒ permet d'ajouter des labels aux lignes et aux colonnes pour accéder aux éléments de façon plus explicite.
 - L'exemple de base, c'est la table excel type BD SQL avec un individu (objet) par ligne.
 - ⇒ Permet de faire l'équivalent du SQL : jointure, group by et pivot.

Notebook

- On mettra nos exemples dans des notebooks.
- Tous les codes de ce chapitre sont dans un notebook dont l'image HTML est ici : <http://bliaudet.free.fr/notebook/NoteBook-Numpy-0.html>

Installation et utilisation d'un notebook

- Commencez par faire l'installation de numpy (pip ou pip3) :

```
pip3 install numpy
```

ou tout d'un coup :

```
pip3 install numpy matplotlib pandas
```

- Installation pip ou pip3

```
pip3 install notebook.
```

- Démarrage du serveur jupyter pour l'utilisation d'un notebook

```
C:> jupyter notebook
```

- Une fois le serveur jupyter démarré, on a un client notebook qui s'ouvre :

```
http://localhost:8888/tree
```

⇒ Nouveau pour ouvrir un notebook

Vitesse de calcul

- On teste les codes dans un notebook qui facilite l'usage du `timeit`.

```
import numpy as np
L = list(range(1000))
a = np.array(L)

# python classique avec compréhension
%timeit [x**2 for x in L]      # 300 micro-sec

# python avec un array numpy
%timeit [x**2 for x in a]     # un peu plus rapide : 200

# "vectorisation" de l'array numpy (opérateur **)
%timeit a**2                  # beaucoup plus rapide : 1,5
```

- `a` c'est un tableau (array) numpy.
- `a**2` : on met le tableau au carré. C'est de la « vectorisation ». C'est la même chose que l'écriture en compréhension, mais c'est plus rapide.
- `timeit` est une fonction python pour calculer le temps d'exécution :
 - ⇒ On écrit `%timeit` dans les notebooks (pas dans l'interpréteur python).
 - ⇒ On y reviendra dans le chapitre sur la vectorisation.

Résultats :

```
284 µs ± 9.77 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
248 µs ± 6.83 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
1.45 µs ± 130 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

pandas

Notebook

- On mettra nos exemples dans des notebooks.
- Tous les codes de ce chapitre sont dans un notebook dont l'image HTML est ici : <http://bliaudet.free.fr/notebook/NoteBook-Pandas-0.html>

Installation

- Commencez par faire l'installation de pandas (pip ou pip3) :

```
pip3 install pandas
```

ou tout d'un coup :

```
pip3 install numpy matplotlib pandas
```

Affichage d'un tableau CSV : équivalent d'un SELECT *

- On peut récupérer le fichier titanic sur github : <https://github.com/mwaskom/seaborn-data>
⇒ Le Titanic, le « hello world » de la data-analyse en python !
- Le fichier csv directement -> [ici](#). Il faut le copier-coller et l'enregistrer dans titanic.csv par défaut dans le dossier de démarrage du serveur jupyter.
- C'est un tableau d'environ 800 lignes.
⇒ On le charge avec la méthode read_csv()
⇒ On affiche les premières lignes avec la méthode head()

```
import pandas as pd

datas = pd.read_csv('titanic.csv')

datas.head()
```

Résultats :

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

Forme ou format d'un tableau : nombre de lignes et de colonnes

```
datas.shape # (891, 15)
```

Sélection de colonnes : équivalent d'un SELECT pclass, survived, sex, age

- On veut projeter uniquement les colonnes pclass, survived, sex et age.
- SELECT SQL :

```
SELECT pclass, survived, sex, age
FROM datas
```

- Code pandas :

```
datas[['pclass', 'survived', 'sex', 'age']]
```

⇒ Notez les doubles crochets.

GROUP BY : taux de survie par sexe et par classe

- On veut calculer le taux de survie des passagers par sexe et par classe
- SELECT SQL :

```
SELECT sex, pclass, avg(survived) as taux_survie, avg(age)
age_moyen
FROM datas
GROUP BY sex, pclass
```

- Code pandas :

```
res=datas[['sex', 'pclass', 'survived', 'age']]\
    .groupby(['sex', 'pclass'])\
    .mean()
res
```

⇒ Le \ permet de passer le code à la ligne, on peut mettre des indentations : c'est plus lisible

- Résultats :

		survived	age
sex	pclass		
female	1	0.968085	34.611765
	2	0.921053	28.722973
	3	0.500000	21.750000
male	1	0.368852	21.750000
	2	0.157407	21.750000
	3	0.135447	26.507589

GROUP BY : statistiques diverses

- **SELECT SQL :**

```
SELECT sex, pclass,  
       avg(survived) as taux_survie,  
       avg(age) age_moyen, min(age) age_min, max(age) age_max  
FROM datas  
GROUP BY sex, pclass
```

- **Code pandas :**

```
res=datas[['sex', 'pclass', 'survived', 'age']]\  
      .groupby(['sex', 'pclass'])\  
      .agg({  
          'survived': ['mean'],  
          'age': ['mean', 'min', 'max']  
      })  
res
```

- ⇒ Le \ permet de passer le code à la ligne, on peut mettre des indentations : c'est plus lisible
C'est utile pour séparer les « .appelDeFonction() ».
- ⇒ On peut passer le code entre parenthèses ou crochets à la ligne : c'est plus lisible
- ⇒ Au lieu d'appliquer la méthode de statistique après le groupby, on applique la méthode agg() qui va nous permettre de passer plusieurs méthodes statistiques.
- ⇒ On passe un dictionnaire dans la fonction agg()

GROUP BY : statistiques diverses et chatgpt

- SELECT SQL :

```
-- nombre et taux de survivants et de non survivants par classe.
select
  class,
  sum(survived) as nb_survivants,
  round(sum(survived)/count(*)*100, 2) as taux_survivants,
  sum(case
    when survived = 0 then 1
    else 0
  end
) as nb_non_survivants,
round(
  sum(case
    when survived = 0 then 1
    else 0
  end
)/count(*)*100, 2
) as taux_non_survivants
from voyageurs
group by class
order by class;
```

- On demande à chatgpt d'écrire la requête en Pandas :

```
result = datas.groupby('class').agg(
  nb_survivants=('survived', 'sum'),
  nb_non_survivants=('survived', lambda x: (x == 0).sum()),
  taux_survivants=('survived', lambda x: round(x.sum() / len(x)
* 100, 2)),
  taux_non_survivants=('survived', lambda x: round((x ==
0).sum() / len(x) * 100, 2))
).reset_index()
result
```

⇒ Ça marche très bien et le code est très propre ! (sur la base d'un code SQL propre aussi !

- Résultats :

	class	nb_survivants	nb_non_survivants	taux_survivants	taux_non_survivants
0	First	136	80	62.96	37.04
1	Second	87	97	47.28	52.72
2	Third	119	372	24.24	75.76

PIVOT

- La fonction `pivot_table()` permet de produire un tableau croisé (qu'on appelle pivot).
- Le pivot n'est pas standard en SQL : on le trouve sous ORACLE et SQL_SERVER mais pas sur MySQL.
- Exemple : un tableau avec :
 - ⇒ en lignes les classes
 - ⇒ en colonnes le sexe
 - ⇒ en données les taux de survie.

	survived	
	female	male
pclass		
1	0.968085	0.368852
2	0.921053	0.157407
3	0.500000	0.135447

- ⇒ 96% des femmes de 1^{ère} classe ont survécu,
- ⇒ 13% des hommes de 3^{ème} classe ont survécu,
- ⇒ etc.

- Code pandas :

```
res=datas[['pclass', 'survived', 'sex']]\n      .pivot_table(index = 'pclass', columns = 'sex')\nres
```

- ⇒ Le `\` permet de passer le code à la ligne, on peut mettre des indentations : c'est plus lisible
C'est utile pour séparer les « `.appelDeFonction()` ».

Autres usages :

- Pour lister les attributs du tableau :

```
datas.columns
```

- Pour avoir une synthèse statistique des attributs numériques :

```
datas.describe()
```

- Pour avoir la moyenne par sexe et classe de tous les attributs numériques :

```
data.groupby(['sex', 'pclass']).mean()
```

- Pour créer un nouveau tableau avec des colonnes en moins :

```
datas2 = datas.drop([
    'sibsp', 'parch', 'fare', 'embarked', 'class', 'who',
    'adult_male', 'deck', 'embark_town', 'alive', 'alone'
],
axis=1
)

datas2.columns
```

⇒ On peut passer le code entre parenthèses ou crochet à la ligne : c'est plus lisible

- Pour créer un nouveau tableau avec des colonnes au choix :

```
datas3=datas[['pclass', 'survived', 'sex', 'age']]

datas3.columns
```


Variante seaborn du PIVOT

- Avec seaborn et la fonction `load_dataset()`, on récupère directement des données de [github](#).
- Ici, on va récupérer le fichier titanic
- La fonction PIVOT est celle de pandas. Mais ici on précise l'attribut de statistique (survived) et la fonction de statistique (mean).
- Code pandas :

```
import numpy as np
import pandas as pd
import seaborn as sns

datas = sns.load_dataset('titanic') # csv de gitub
datas.columns
datas.head()
res = datas.pivot_table(
    'survived',
    aggfunc=np.mean,
    index='class',
    columns='sex'
)
res
```

⇒ On peut passer le code entre parenthèses ou crochets à la ligne : c'est plus lisible

- Résultats :

sex	female	male
class		
First	0.968085	0.368852
Second	0.921053	0.157407
Third	0.500000	0.135447

Notebook

- On mettra nos exemples dans des notebooks.
- Tous les codes de ce chapitre sont dans un notebook dont l'image HTML est ici : <http://bliaudet.free.fr/notebook/NoteBook-Matplotlib-0.html>

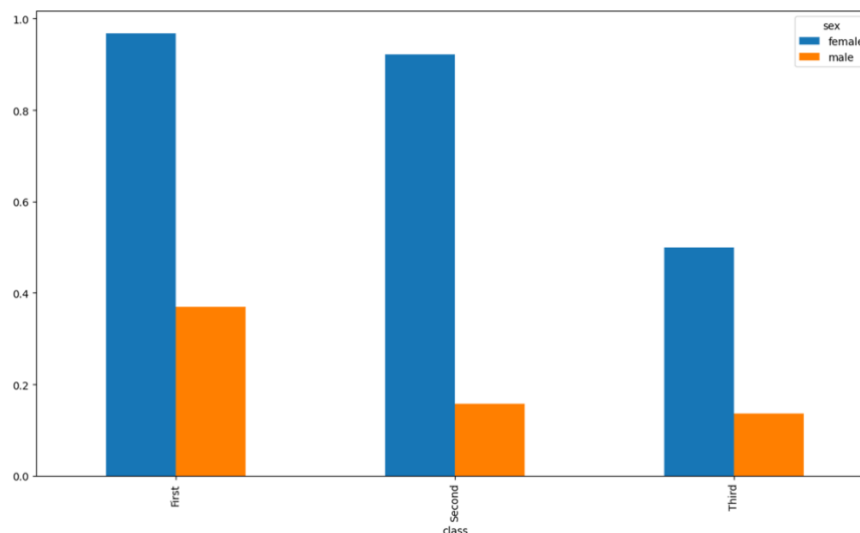
Notebook

- Pour avoir un graphique, on utilise la fonction « plot » de la bibliothèque « matplotlib ».
- Le code est le suivant, appliqué au tableau « res » précédemment calculé, par pivot ou par group by.
- Code :

```
import matplotlib.pyplot as plt

res.plot(kind='bar') # pour avoir des barres
plt.show()
```

- Résultat :



- Variante :
⇒ On peut sommer le nombre de survivant par sexe et par classe plutôt que calculer un pourcentage.

2 - NUMPY

Présentation (5p - 27-31)

Références officielles

- <https://numpy.org/>
- <https://numpy.org/doc/stable/>
- <https://numpy.org/doc/stable/reference/index.html#reference>

Webographie

- Plein de ressources sur internet.
- **Cours MOOC**
 - <https://lms.fun-mooc.fr/courses/course-v1:UCA+107001+session02/courseware/82ac208ce93b479f90a5a65e04753f5a/d2f18e116d8244b6a652fc17fe1fe4e2/>
- **Exos MOOC**
 - <https://lms.fun-mooc.fr/courses/course-v1:UCA+107001+session02/courseware/82ac208ce93b479f90a5a65e04753f5a/4570db6c9e784167b2e741a3e9c063ac/>
- **Cours OCR**
 - <https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/5560976-familiarisez-vous-avec-lecosysteme-python>
 - <https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science/4740941-plongez-en-detail-dans-la-librairie-numpy>
- **Cours W3School**
 - <https://www.w3schools.com/python/numpy/default.asp>
 - <https://www.w3schools.com/python/default.asp>

Installation pip ou pip3

- Installation un par un :
 - ⇒ pip3 install numpy
 - ⇒ pip3 install matplotlib
- Installation tout d'un coup :
 - ⇒ pip3 install numpy matplotlib pandas
- <https://pypi.org> : c'est la plateforme qui distribue les bibliothèques non standards via l'outil pip ou pip3.

Utilisation de numpy

- Le bon usage :

```
import numpy as np
```

- On peut aussi écrire :

```
import numpy          # à éviter
```

Rappels introductifs

Origine

- Python 1.0 : 1994 : langage ancien et mature
 - NumPy 1.0 : 2006 : bibliothèque plus récente
 - Pandas 1.0 : 2008 : bibliothèque encore récente
- ⇒ NumPy et Pandas sont beaucoup plus récents.

Philosophie

- Python : simplicité et souplesse (versus une certaine efficacité = optimisation).
- NumPy et Pandas : efficacité versus simplicité et souplesse.

Principes

- **Numpy :**
 - ⇒ pour des tableau python, multidimensionnel, avec des objets de même type.
 - L'exemple de base, c'est le tableau à 1 ou 2 dimensions avec des entiers, des réels ou des booléens.
 - ⇒ rapide grâce à la vectorisation.
- **Pandas :**
 - ⇒ permet d'ajouter des labels aux lignes et au colonnes pour accéder aux éléments de façon plus explicite.
 - L'exemple de base, c'est la table excel de BD avec un individu (objet) par ligne.
 - ⇒ Permet de faire l'équivalent du SQL : jointure, group by et pivot.

Bilan

- Ces librairies ont leurs limites et leurs lourdeurs, mais c'est ce qui se fait de mieux.
- ⇒ Il faut faire avec et profiter pleinement de la puissance de ces librairies.

4 notions fondamentales numpy

- Numpy est la librairie qui permet de manipuler des **tableaux multidimensionnels** qu'on appelle **ndarray** (n dimensionnal array).
 - ⇒ Tableaux à 1 dimension (comme des listes classiques)
 - ⇒ Tableaux à 2 dimensions (comme des tableaux excel type BD classiques)
 - ⇒ Tableaux à 3 dimensions (pour se représenter un volume)
 - ⇒ Tableaux à 4 dimensions (pour l'analyse multi-dimensionnelle sur un nombre quelconque de variables : difficile !)
- Il y a **4 notions fondamentales en numpy** qui tournent **autour du ndarray** :

Ndarray
Indexation avancée
Vectorisation
Broadcasting

- Ces notions sont... fondamentales !
 - ⇒ **Il faut se familiariser avec pour penser NumPy.**
 - ⇒ Elles relèvent de la programmation fonctionnelle.

Programmation fonctionnelle

- Travailler en numpy consiste à **penser en programmation fonctionnelle** au sens large.
- La programmation fonctionnelle consiste essentiellement à **manipuler des tableaux** (listes, collection, matrices, etc.) **sans faire de boucles !**
- La programmation fonctionnelle, au sens large c'est :
 - ⇒ Le langage Lisp (vintage)
 - ⇒ Le langage MatLab
 - ⇒ La programmation fonctionnelle en JavaScript, Python, Java et les lambda expression
 - ⇒ La compréhension de liste en Python
 - ⇒ Le SQL
 - ⇒ Le slicing, le broadcasting et la vectorisation en NumPy
 - ⇒ Pandas

Le type ndarray (25p - 32-56)

- On mettra nos exemples dans des notebooks.
- Tous les codes de ce chapitre sont dans un notebook dont l'image HTML est ici : <http://bliaudet.free.fr/notebook/NoteBook-Numpy-2-ndarray.html>

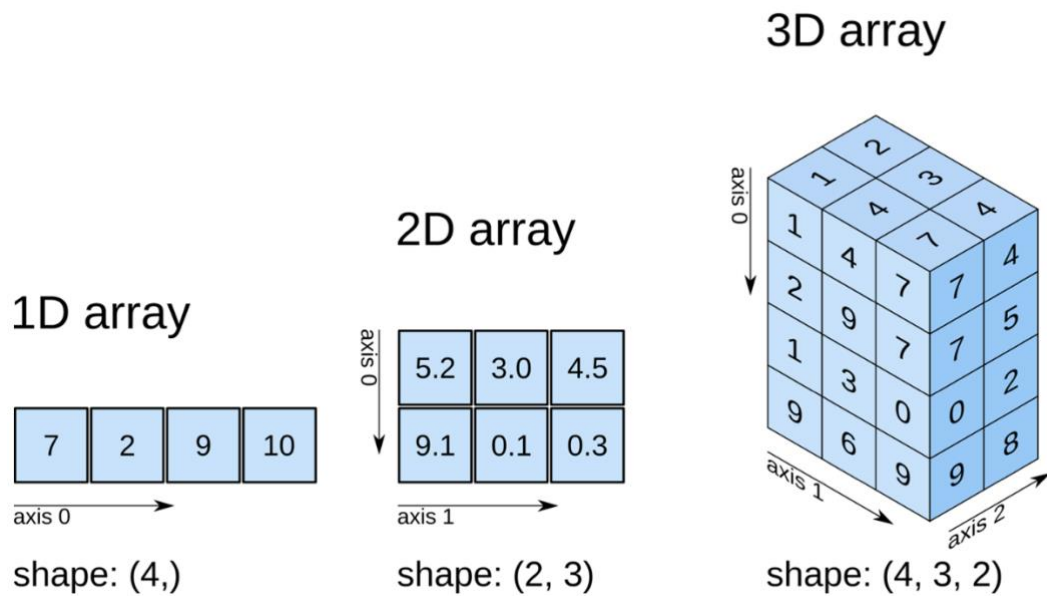
Qu'est-ce que c'est ?

Principes

- Un tableau numpy est de type **ndarray**
- On parle de **tableau numpy** ou de ndarray (plutôt tableau numpy).
- C'est une **zone de mémoire contigüe qui en contient que des éléments du même type**.
 - ⇒ Comme la mémoire est contigüe, l'accès aux données peut être très rapide.
 - ⇒ Pour chaque type de donnée, on aura des fonctions spécialisées.
- **On peut changer les valeurs d'un tableau**, à condition de garder le même type qu'on appelle « **dtype** » (data type : un type python ou un type équivalent numpy. Par exemple : int64).
 - ⇒ En **python**, un tableau est une liste qui peut contenir **n'importe quel type de données**.
 - ⇒ En **numpy**, c'est un tableau d'un **type fixe**.
 - Numpy interprète les chaînes correspondant à un nombre comme un nombre si nécessaire et réciproquement !
- **On ne peut pas changer la taille d'un tableau** : ajouter ou supprimer des éléments. La taille du tableau numpy est fixe.
 - ⇒ En **python**, un tableau est une liste dans laquelle **on peut ajouter de nouveaux éléments**.
 - ⇒ En **numpy**, la **taille est fixe**.

Schémas

https://training.digitalearthfrance.org/fr/latest/python_basics/02_numpy.html



		Vecteur	Matrice	Pavé
Type	dtype	int64	float64	int64
Dimension	ndim	1	2	3
Format	shape	(4,)	(2, 3)	(4, 3, 2)
Nombre d'éléments	size	4	6	24
Taille en octets	nbytes	32	48	192
Taille d'un élément en octet	itemsize	8	8	8

Documentation officielle

- C'est théorique, mais on trouve des listes d'attributs et de méthodes accessibles et des exemples.

⇒ <https://numpy.org/doc/stable/reference/arrays.html>

⇒ <https://numpy.org/doc/stable/reference/arrays.ndarray.html>

Méthodes de manipulation

- Numpy fournit des **attributs d'informations** sur le tableau complet : nbytes, size, etc.
- Numpy fournit des **fonctions de création** d'un tableau numpy : array(), ones(), etc.
- Numpy fournit des **fonctions spécialisées pour chaque type d'objet** qu'on peut avoir dans tableau numpy.

Premières fonctions de création d'un tableau numpy

Fonction array()

<https://numpy.org/doc/stable/reference/generated/numpy.array.html?highlight=array#numpy.array>

➤ Principes

- La fonction array transforme une liste python en tableau numpy.

```
import numpy as np

L = list(range(1000))
a = np.array(L)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
```

- On renomme numpy en « np » à l'importation : c'est l'usage.
- On ne peut pas ajouter ou supprimer un élément dans un tableau numpy.
- Les éléments ont tous le même type qu'on appelle dtype.

➤ Variantes :

- Notez la syntaxe : la présence ou pas de []

```
tab=np.array( list(range(5)) )

tab=np.array( range(5) )

tab=np.array( [ i for i in range(5) ] )

tab=np.array( [ i%2 for i in range(5) ] )
```

Forme, dimension et taille d'un tableau : shape, ndim, size

```
cube=np.ones( (4, 3, 2) )
cube.shape    # (4, 3, 2)
cube.ndim     # 3
cube.size     # 24
```

Fonction ones(), zeros(), full()

<https://numpy.org/doc/stable/reference/generated/numpy.ones.html>

- La fonction **ones()** permet de créer un tableau numpy (ndarray) rempli de 1.
- La fonction **zeros()** permet de créer un tableau numpy (ndarray) rempli de 0.
- La fonction **full()** permet de créer un tableau numpy (ndarray) rempli de ce qu'on veut.

```
tableau=np.ones(4)
matrice=np.ones((3,2))
cube=np.ones((4,3,2))
```

- On peut utiliser l'argument « shape » mais c'est facultatif :

```
tableau=np.ones(shape=4)
tableau=np.ones(shape=(4,3))
```

⇒ A noter que (4,3) est un tuple python : on définit les dimensions du tableau.

- On peut modifier le contenu du tableau :

```
matrice[1][1]=5
matrice[1, 0]=9 # mieux vaut utiliser cette écriture
```

- Exemples zeros() et full()

```
a1=np.zeros(5)
a2=np.full(5, 'coucou')
```

Fonction random.randint()

<https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html>

- Création d'un tableau de 3 lignes et 3 colonnes rempli aléatoirement avec des entiers de 1 à 10 :

```
a = np.random.randint(1, 10, (3, 3))  
array([[7, 8, 5, 7],  
       [5, 3, 1, 7],  
       [1, 6, 2, 7],  
       [5, 4, 1, 9]])
```

- On peut utiliser l'argument « size » mais c'est facultatif :

```
a = np.random.randint(1, 10, size=(3, 3))
```

⇒ La « size » c'est la même chose que le « shape » dans la fonction ones() : ce n'est pas très pratique. C'est un manque de maturité du langage.

Fonction random.random()

<https://numpy.org/doc/stable/reference/random/generated/numpy.random.random.html>

- Création d'un tableau de 10 éléments entre 0 et 1 :

```
a = np.random.random(10)  
array([0.09191657, 0.63108598, 0.21663367, 0.47830876, 0.32190306,  
       0.22137602, 0.88724274, 0.94865523, 0.86640787, 0.75000379])
```

Fonction arange()

<https://numpy.org/doc/stable/reference/generated/numpy.arange.html?highlight=arange#numpy.arange>

- La fonction arange() permet de créer un tableau numpy (ndarray) rempli de 0 à N

```
a=np.arange(10)
a # array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- La fonction arange(début, fin, intervalle) permet de remplir un tableau numpy (ndarray) de début jusqu'à fin non compris en ajoutant des nombres à intervalle fourni régulier (comme un slice). L'intervale peut être un réel.

```
a=np.arange(1, 11, 2)
a # array([1, 3, 5, 7, 9]) : pas de 11

b=np.arange(5, 6, 0.25)
b # array([5. , 5.25, 5.5 , 5.75]) : pas de 6
```

Fonction eye()

- La fonction eye() permet de créer une matrice carrée numpy (ndarray) rempli de 0 avec des 1 en diagonale.
- Par défaut, ce sont des float64

```
a=np.eye(3)

b=np.eye(3, dtype=int)
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

Fonction empty()

- Création d'un tableau vide : c'est le plus rapide. Les valeurs initiales sont aléatoires (c'est l'état de la mémoire) même si souvent égales à 0.

```
np.empty((3, 3))  
array([[ 2.60605835e-31, -5.21211670e-31,  1.30302917e-31],  
       [-5.21211670e-31,  1.13363538e-30, -3.51817877e-31],  
       [ 1.30302917e-31, -3.51817877e-31,  2.01969522e-31]])
```

- En précisant le type, on pourra n'avoir que des entiers :

```
np.empty((5, 5), dtype=np.int8 )  
array([[ 0,  0,  0,  0,  0],  
       [ 0,  0, 32, -71, 115],  
       [ 4, 104, -1,  7,  0],  
       [48,  2,  0, 61, 49],  
       [ 0,  0,  0,  0,  0]], dtype=int8)
```

Fonction copy()

- La fonction copy permet de dupliquer un tableau numpy.

```
b = np.copy(a)
```

Fonction reshape()

- La fonction reshape permet de réorganiser un tableau numpy en changeant son nombre de lignes et de colonnes :

⇒ Attention, on doit avoir le même nombre d'éléments au début et à la fin.

```
tab = np.arange(12)
# array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

# On remplit en partant de l'indice de droite

mat1 = tab.reshape(3, 4)
# mat 1 = np.reshape(tab, (3,4))
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])

pave1 = tab.reshape(2, 3, 2)
# array([[[ 0,  1],
#         [ 2,  3],
#         [ 4,  5]],
#        [[ 6,  7],
#         [ 8,  9],
#         [10, 11]]])

pave2 = mat1.reshape(3, 2, 2)
# array([[[ 0,  1],
#         [ 2,  3]],
#        [[ 4,  5],
#         [ 6,  7]],
#        [[ 8,  9],
#         [10, 11]]])

pave3 = pave2.reshape(2, 2, 3)
# array([[[ 0,  1,  2],
#         [ 3,  4,  5]],
#        [[ 6,  7,  8],
#         [ 9, 10, 11]]])
```


Exercices

- Dans ces exercices, le but est de se rappeler les principes de la manipulation de matrices et de volumes en programmation python classique.

Matrice

- On veut remplir une matrice de 3 lignes et 2 colonnes avec des valeurs de 1 à n en la remplissant comme avec un `arange()`, ligne par ligne.
- Pour cet exercice, on utilisera obligatoirement les simples boucles python : `for i in range`.
- On fera en sorte d'avoir un code qui permette de changer de côté facilement la taille de la matrice.
- Faites des tests en changeant le format de la matrice.

⇒ Exemple d'affichage pour une matrice (3,2) :

```
0 0 : 1
0 1 : 2
1 0 : 3
1 1 : 4
2 0 : 5
2 1 : 6
```

⇒ Solution : <http://bliaudet.free.fr/IMG/txt/arangeMatrice.py>

Cube.

- On veut remplir un cube de côté 2 avec des valeurs de 1 à n en l remplissant progressivement niveau par niveau et ligne par ligne dans chaque niveau.
- Pour cet exercice, on utilisera obligatoirement les simples boucles python : `for i in range`.
- On fera en sorte d'avoir un code qui permette de changer de côté facilement la taille du volume.
- Faites des tests en changeant le format du volume.

⇒ Exemple d'affichage pour un cube (2,2,2) :

```
0 0 0 : 1
0 1 0 : 2
1 0 0 : 3
1 1 0 : 4
0 0 1 : 5
0 1 1 : 6
1 0 1 : 7
1 1 1 : 8
```

⇒ Solution : <http://bliaudet.free.fr/IMG/txt/arangeCube.py>

Première approche de la programmation vectorielle

Variantes algorithmiques

- *Soit la fonction « maf » et le tableau de valeurs de X*

```
def maf(x):  
    return 4*x**2 - 2*x + (1 if x <= 0 else 10)  
  
X=np.arange(-10, +10)
```

- *On veut calculer Y pour toutes les valeurs de X. On a 3 techniques :*

- ⇒ Boucles d'algorithmique classique
- ⇒ Compréhension de liste ou générateur
- ⇒ Vectorisation

- *En algorithmique classique, on écrit :*

```
Y = []  
for x in X:  
    Y.append(maf(x))
```

- *En compréhension de liste, on écrit :*

```
Y = [maf(x) for x in X]
```

- ⇒ Avec un générateur, on écrit (le générateur économise la mémoire)

```
Y = (maf(x) for x in X)
```

- *En numpy, on doit penser encore plus court, grâce à la vectorisation :*

```
Y = maf(X)
```

- *Rappels python sur la compréhension de liste et les générateurs :*

- Page 15, page 46, pages 104-115 : chapitre 51
- <http://bliaudet.free.fr/IMG/pdf/Python.pdf>

Types, ndarray et dtype

rappels sur les types python

⇒ <https://docs.python.org/fr/3.5/library/stdtypes.html#>

⇒ https://www.w3schools.com/python/numpy/numpy_data_types.asp

types numpy

⇒ <https://numpy.org/doc/stable/user/basics.types.html>

- On peut retenir :
⇒ bool, int, uint (entier non signé), float et complex ;

ndarray

- Un tableau numpy est de type ndarray
⇒ <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html?highlight=ndarray#numpy.ndarray>

```
import numpy as np
a = np.ones(4)

type(a)      # numpy.ndarray
```

dtypes

- Les types d'un tableau numpy s'appellent « dtype ».
⇒ <https://numpy.org/doc/stable/reference/arrays.dtypes.html>
- On distingue :
 - ⇒ les types « scalaires » (une valeur unique)
 - ⇒ les types « composites » (plusieurs valeurs : dictionnaire)

```
import numpy as np
a = np.ones(4)

a          # array([1., 1., 1., 1.])
type(a)    # numpy.ndarray
a.dtype    # dtype('float64')
type(a[0]) # numpy.float64
a[0].dtype # dtype('float64')
```

⇒ numpy.float64, c'est un « dtype ».

Les types scalaires

liste des types scalaires :

```
np.sctypes
```

- Résultats :

```
{'int': [numpy.int8, numpy.int16, numpy.int32, numpy.int64],  
 'uint': [numpy.uint8, numpy.uint16, numpy.uint32, numpy.uint64],  
 'float': [numpy.float16, numpy.float32, numpy.float64,  
           numpy.float128],  
 'complex': [numpy.complex64, numpy.complex128, numpy.complex256],  
 'others': [bool, object, bytes, str, numpy.void]}
```

⇒ **Les numériques** : int, uint, float, complex (au sens arithmétique) et autres.

⇒ **Les autres** : others sont les autres types scalaires

Exemple : création d'un tableau numpy d'un type scalaire numérique

- Pour avoir un tableau numpy d'entiers sur 8 bit :

```
tab=np.array([1, 5, 100], dtype=np.int8)  
  
tab.dtype      # dtype('int8') : type des éléments du tableau  
tab.nbytes     # 3 : taille du tableau en byte (octet)  
tab.itemsize   # 1 : taille d'un élément en byte (octet)  
tab[0].nbytes  # 1 : taille de l'élément [0] en byte (octet)  
  
tab = np.array([1, 2, 4, 8j]) # 8j : écriture complexe  
tab.dtype      # dtype('complex128') : 16*8=128  
tab.nbytes     # 64 : taille du tableau en byte (octet): 16*4=64  
tab.itemsize   # 16 : taille d'un élément en byte : 16*8=128  
tab[0].nbytes  # 16 : taille de l'élément [0] en byte.
```

Transtypage automatique et limites

- Si on écrit :

```
tab=np.array([1, 5.5, 128], dtype=np.int8)
```

⇒ Le 5.5 devient un 5 car on a un int8

⇒ Le 128 devient -128 car on est sorti des limites d'un int8 qui va de -128 à 127.

nan : not a number : valeur invalide

- np.nan est de type float. Il n'y a pas de nan en entier.

- Si on écrit :

```
tab=np.array([1, 5.5, np.nan], dtype=np.int8)
```

⇒ Ça bogue ! nan est un float.

- Si on écrit :

```
tab=np.array([1, 5.5, np.nan], dtype=np.float16)
```

⇒ Ça passe.

⇒ En datascience, il est classique d'avoir des valeurs invalides.

⇒ Attention, ça n'est possible qu'avec le type float.

Principe : éviter les conversions implicites !

- Si on écrit :

```
tab=np.array([1, 5, np.nan])
tab.dtype #
```

⇒ tab est un tableau de float : car nan est un float.

- Si on écrit :

```
tab=np.array([1, 5.5, np.nan], dtype=np.int8)
```

⇒ Ça bogue : le type de np.nan est un float : ce n'est pas un int8.

⇒ Il vaut mieux éviter les conversions implicites et toujours préciser le type des tableaux numpy avec l'argument dtype.

Changement de type - transtypage

```
a = np.ones(100)
a.dtype          # dtype('float64')

a=a.astype(int)   # pour passer en int
a.dtype          # dtype('int64')

a=a.astype(np.int8) # pour passer en int
a.dtype          # dtype('int8')

a=a.astype(np.float16) # pour passer en int
a.dtype          # dtype('float16')

a=a.astype(np.bool_) # pour passer en int
a.dtype          # dtype('bool')
```

- Tous les types :

⇒ <https://numpy.org/doc/stable/reference/arrays.scalars.html?highlight=float64#numpy.float64>

Rappels de transtypage python

```
a = 5
type(a) # int
a = float(a)
type(a) # float
a = (a, )
type(a) # tuple
```

Les autres types scalaires

Présentation

- Liste des types scalaires :

```
np.sctypes
```

⇒ int, uint, float, complex (au sens arithmétique) et autres.

⇒ others sont les autres types scalaires

Les autres types scalaires

```
np.sctypes['others']
```

⇒ [bool, object, bytes, str, numpy.void]

- **Les booléens** fonctionneront comme des scalaires.
- **Les tableau d'object** contiennent uniquement des références vers des objets de taille variables. Ils sont **peu utiles** pour le calcul car ça engendre une forte perte de performance.
- **numpy.void** permet de stocker des objets de taille fixe et de type quelconque. Ainsi on retrouve la performance.
- **str et bytes : cf chapitre suivant.**

Les types str et bytes

- Avec **bytes** et **str**, on pourrait croire qu'on peut mettre des « str » ou des « bytes » de taille quelconque dans un tableau numpy :

⇒ Ce n'est pas le cas ! **La taille des objets d'un tableau numpy est toujours fixe !**

```
tab = np.array(['spam', 'bean'], dtype=str)

tab.nbytes      # 32 : 4*4 = 16 - 16*2 = 32
tab.dtype       # dtype('<U4')
```

⇒ On n'écrit plus np.str mais str directement

⇒ Le type implicite est « U4 » : de l'unicode (UTF) sur 4 caractères : c'est la taille max des str dans le tableau. U4 car la taille maximum des string données est de 4.

⇒ tab.nbytes vaut 32 : chaque caractère est codé sur 4 bytes (4*4=16 et 16*2=32).

- Si on écrit :

```
tab = np.array(['spam', 'beans'], dtype=str)

tab.dtype       # dtype('<U5') : c'est le type implicite
tab.nbytes      # 40 : 5*4 = 20 - 20*2 = 40

tab[0]          # 'spa'
tab[0].dtype    # dtype('<U4')
tab[0].nbytes   # 16 : 4*4 = 16
```

- Typage explicite : à privilégier

```
tab = np.array(['spam', 'beans'], dtype=(str, 3))

tab.nbytes      # 24 : 3*4=12 et 12*2=24.
tab.dtype       # dtype('<U3')

tab[0]          # 'spa'
tab[0].dtype    # dtype('<U3')
tab[0].nbytes   # 12 : 4*3 = 12
```

- bytes

⇒ Le byte c'est 1 octet

```
tab = np.array(['spam', 'beans', 100], dtype=bytes)

tab.dtype      # dtype('S5') : le plus grand en implicite
tab.nbytes     # 15 : 5*3

tab[0]         # b'spam'
tab[0].dtype   # dtype('S4')
tab[0].nbytes  # 4 :

tab[2]         # b'100'
tab[2].dtype   # dtype('S3')
tab[2].nbytes  # 3 : c'est traité comme une string

tab[2]*2       # b'100100'

tab = np.array(['spam', 'beans', 100], dtype=bytes)
```

```
tab = np.array(['spém', 'beans'], dtype=bytes)
```

⇒ Ca bogue ! ord('è') vaut 233 : au-dessus de la limite à 128

⇒ chr(97) vaut 'a', chr(65) vaut 'A'

```
tab = np.array(['spam', 'beans', 100], dtype=(bytes, 2))

tab.dtype      # dtype('S2') : taille explicite
tab.nbytes     # 6 : 2*3

tab[0]         # b'sp'
tab[0].dtype   # dtype('S2')
tab[0].nbytes  # 2

tab[2]         # b'10'
tab[2].dtype   # dtype('S2')
tab[2].nbytes  # 2 : c'est traité comme une string
```

Les types composites

- On peut mettre des types composites dans les tableaux numpy :
⇒ Par exemple : un dictionnaire.
- Ce n'est pas utilisé car ça ne permet pas de faire des calculs efficaces.
⇒ C'était déjà le cas avec des références vers les objets.
- Exemple avec d'un tableau numpy de dictionnaires

```
tab.dtype      # dtype('O') : type des éléments du tableau
tab.nbytes     # 24 : taille du tableau en byte (octet)
tab.itemsize   # 8 : taille d'un élément en byte (octet)

#tab[0].nbytes # error : 'dict' object has no attribute 'nbytes'
#tab[0].dtype  # error : 'dict' object has no attribute 'dtype'
type(tab[0])   # dict
tab[2]['c'].    # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Attributs de base : size, shape, ndim, itemsize

```
tab = np.full(3, 5, int)          # [5, 5, 5]

tab.size                          # nombre total d'elt : 3
tab.shape                        # (3, )
tab.ndim                         # 1
tab.itemsize                     # 8: nombre d'octets par elt
                                # int64 par défaut pour int

cube=np.ones((2,3,4))            # [ [ [1., 1., 1., 1.],
                                #      [1., 1., 1., 1.],
                                #      [1., 1., 1., 1.] ],
                                #
                                #      [ [1., 1., 1., 1.],
                                #      [1., 1., 1., 1.],
                                #      [1., 1., 1., 1.] ] ]

cube.size                        # nombre total d'elt : 24
cube.shape                      # (4, 3, 2)
cube.ndim                       # 3
cube.itemsize                   # 8: nombre d'octets par elt
                                # float64 par défaut
```

Fonctions de création de base :

```
np.array([1,2, 3, 4, 5])           # [1,2, 3, 4, 5])

np.array([[1,2, 3], [4, 5, 6]])     # [[1, 2, 3],
                                     #  [4, 5, 6]]

np.array(range(5))                  # [0, 1, 2, 3, 4]
np.arange(5)                        # [0, 1, 2, 3, 4]
np.arange(2, 5)                     # [2, 3, 4]

cube=np.eye(3)                      # [ [ [1., 0., 0.],
                                     #    [0., 1., 0.],
                                     #    [0., 0., 1.] ],

np.array( [i**2 for i in range(5)]) # [ 0,  1,  4,  9, 16]

np.ones(5)                          # [1., 1., 1., 1., 1.]
np.zeros(5)                         # [0., 0., 0., 0., 0.]
np.full(3, 5)                       # [5, 5, 5]
tab = np.full(3, 'yes')              # ['yes', 'yes', 'yes']

cube=np.ones((2,3,4))               # [ [ [1., 1., 1., 1.],
                                     #    [1., 1., 1., 1.],
                                     #    [1., 1., 1., 1.] ],

                                     #  [ [1., 1., 1., 1.],
                                     #    [1., 1., 1., 1.],
                                     #    [1., 1., 1., 1.] ] ]

np.random.randint(1, 10, (3, 3))
np.random.random(10)

np.empty((5))
np.empty((5, 5), dtype=np.int8 )

tab2 = np.copy(tab)

# 5 valeurs régulièrement espacées de 1 à 9
np.linspace(1, 9, 5)                 # [1., 3., 5., 7., 9.]

tab = np.arange(12)
mat1 = tab.reshape(3, 4)
np.reshape(tab, (3,4))
```

Fonctions de base : min, max, sum, mean, quantile, prod

```
tab = np.arange(1, 10)      # [1, 2, 3, 4, 5, 6, 7, 8, 9]
tab.size                    # 9. Attention, éviter len !!!
np.min(tab)                 # 0
np.max(tab)                 # 9
np.sum(tab)                 # 45
np.prod(tab)                # 362_880
np.mean(tab)                # 5.0
np.quantile(tab, 0.5)       # 5.0
np.quantile(tab, 0.75)      # 7.0

tab2 = np.arange(1, 9)
cube=tab2.reshape((2,2,2))
print(cube.shape)           # (2,2,2)
print(cube.size)            # 8. Attention, éviter len !!!
print(np.min(cube))         # 1
print(np.max(cube))         # 8
print(np.sum(cube))         # 36
print(np.prod(cube))        # 40_320
print(np.mean(cube))        # 4.5
print(np.quantile(cube, 0.5)) # 4.5
print(np.quantile(cube, 0.75)) # 6.25
```

Fonctions de base : tri

```
# retourne un tableau trié
np.sort(a)

tri le tableau a
a.sort()
```

Exercices .

Exercice .

Coder les tableaux de la page 29. Affichez-les et affichez les informations présentées dans le tableau récapitulatif.

Exercice .

Créer des tableaux de type int, int32, float, float32, complex
Ces tableaux seront remplis de 0 à 10, de 5 à 10, uniquement de 1, uniquement de 0.
Afficher le format du tableau, le nombre d'éléments du tableau, le type d'un élément, la taille complète du tableau (en bytes), la taille d'un élément (en bytes).

Créer un tableau « a » de 10 entiers multiples de 3 démarrant en 12.
Afficher ce tableau : il y a combien de crochets ?
Afficher la dimension et le format.

Passer le « à la verticale » : il y a combien de crochets ?
Que vaut a[0], a[0,0], a[0,4], a[4,0]
Afficher la dimension et le format.

Reformatez-le en un tableau de 2 lignes et 5 colonnes.
Que vaut a[0], a[0,0], a[0,4], a[4,0]
Afficher la dimension et le format.

Afficher le type du format.

Exercice .

Créer un vecteur de taille 30 (on dit vecteur pour un tableau à 1 dimension) rempli aléatoirement d'entiers compris entre 1 et 100
Calculer la valeur minimum, la valeur maximum et la moyenne.
Faites un tableau de taille 100 et déterminez la valeur de la médiane et du premier quartile.

Faites la même chose avec une matrice 10x10 (on dit matrice pour les tableaux à 2 dimensions).

Sur un tableau de 10, calculer la somme de tous les éléments et le produit de tous les éléments.

Exercice mean, max, reshape .

Somme le pavé suivant :

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]],

      [[13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]])
```

Calculer la somme et la moyenne pour tout le pavé.

La somme et la moyenne pour chaque plan du pavé (la matrice de 1 à 12, et celle de 13 à 24)

La somme et la moyenne pour chaque ligne du pavé (les lignes de 1 à 4, 5, à 8, 9 à 12, 13 à 16, etc.).

Solution : <http://bliaudet.free.fr/IMG/txt/MeanMaxReshape.py>

- On mettra nos exemples dans des notebooks.
- Tous les codes de ce chapitre sont dans un notebook dont l'image HTML est ici : <http://bliaudet.free.fr/notebook/NoteBook-Numpy-2-indexation.html>

Présentation

- On va aborder les points suivants
 - ⇒ Le **slicing** python appliqué aux tableaux numpy va nous permettre d'extraire des valeurs du tableau (comme en SQL) dans des « vues » au sens d'une « vue » SQL.
 - ⇒ Le **reshaping** qui permet de redimensionner les tableaux numpy.
 - ⇒ L'**indexation avancée** va permettre de sélectionner des éléments de manière plus expressives.

➤ *Rappels python sur le slicing :*

- liste[d : f : p] : sous tableau démarrant en d, jusqu'à f-1, par pas de p.
- Par défaut : démarrant en 0, jusqu'à longueur-1, par pas de 1

```
l=[i for i in range(100)]  
l[:10]      # [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]  
l[10:15]    # [10, 11, 12, 13, 14]  
l[10:31:5]  # [10, 15, 20, 25, 30]
```

- §2.3 - Sequences : p.34
 - §2.4 - Les Listes : p.36
 - §3.2 - Les Tuples : p.60
- ⇒ <http://bliaudet.free.fr/IMG/pdf/Python.pdf>

Slicing et vue

Slicing multidimensionnel

- Le slicing permet de définir un **sous-tableau en limitant les lignes et les colonnes**.
 - ⇒ On utilise la syntaxe du slice python : **1 sous tableau démarrant en i, jusqu'à j-1, par pas de 1**
 - ⇒ **Mais c'est doublé : on slice les lignes, puis les colonnes**
 - ⇒ C'est une extraction de données (comme en SQL).

➤ *Exemple :*

```
# creation d'une matrice 3, 3
a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]], dtype='int8')

a
# slicing : on veut sélectionner : 2, 3 (ligne 0, col 1 et 2)
#                                     6, 7 (ligne 1, col 1 et 2)
print (a[0:2, 1:3])
# ou encore :
print (a[:2, 1:3])
```

Slicing et vue

- Le résultat est dans une « **vue** » (comme en SQL) : on ne duplique pas les données.
- C'est un slice comme en python : **1 sous tableau démarrant en i, jusqu'à j-1, par pas de 1**.
- Par défaut : **démarrant en 0, jusqu'à longueur-1, par pas de 1**

➤ *Code :*

```
# creation d'une matrice 3, 3
a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]], dtype='int8')

a
# slicing : que les colonnes paires
vue = a[:, 1::2]
vue
array([[ 2,  4],
       [ 6,  8],
       [10, 12]], dtype=int8)
```

Notion de vue

- Le résultat d'un slicing numpy n'est pas un nouveau tableau mais une « vue » sur le tableau source.
- Si on modifie le contenu du tableau source, la vue est modifiée aussi.

```
a[2,2]=99  
array([[ 1,  2,  3,  4],  
       [ 5,  6,  7,  8],  
       [ 9, 10, 99, 12]], dtype=int8)
```

```
vue  
array([[ 7,  8],  
       [99, 12]], dtype=int8)
```

⇒ Avec les tableaux numpy, on a une zone continue de mémoire et on travaille sur cette zone continue.

Inverse du slice : concatenate() et _r

- Le slice permet d'extraire un morceau.
- La fonction concatenate() permet de faire un tableau à partir de plusieurs morceaux (de même dimension).
- Le _r[] permet de faire la même chose

⇒ Par exemple : pour passer de :

[1, 2, 3, 4, 5, 6, 7, 8, 9]

⇒ à

[2, 4, 5, 6, 8, 9]

⇒ On peut faire. :

[2] concat [4, 5, 6] concat [8, 9]

```
a=np.arange(1, 10)  
a  
b=np.concatenate([ a[1:2], a[3:6], a[7:] ])  
b  
c=np.r_[ a[1:2], a[3:6], a[7:] ]  
c
```

Slicing de colonnes

- On peut slicer pour récupérer les colonnes d'une matrice.
- Par exemple, cette matrice :

```
a=np.array([[ 0,  1,  2,  3,  4],
            [ 5,  6,  7,  8,  9],
            [10, 11, 12, 13, 14]])
```

➤ *La colonne 3 :*

```
b=a[:,3]
b
array([ 3,  8, 13 ])
```

➤ *Les colonnes 3 et 4 :*

```
b=a[:,3:5]
b
array([[ 3,  4],
       [ 8,  9],
       [13, 14]])
```

➤ *Les colonnes paires :*

```
c=a[:,::2]
c
array([[ 0,  2,  4],
       [ 5,  7,  9],
       [10, 12, 14]])
```

- Le slice de colonnes reste une vue :

```
a[2,2]=99
c
array([[ 0,  2,  4],
       [ 5,  7,  9],
       [10, 99, 14]])
```

Modifier un slicing de colonnes

- On peut modifier les valeurs récupérées par slicing :

```
a[:, ::2] = 0
a
array([[ 0,  1,  0,  3,  0],
       [ 0,  6,  0,  8,  0],
       [ 0, 11,  0, 13,  0]])
```

Extraire un bloc par slicing

- On peut modifier les valeurs récupérées par slicing :

```
bloc = a[1:3, 2:5]
bloc
array([[ 0,  8,  0],
       [ 0, 13,  0]])
```

Modifier un slicing de bloc

```
a=np.arange(15).reshape(3,5)
a[1:3, 2:5]=0
a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  0,  0,  0],
       [10, 11,  0,  0,  0]])
```

Passer d'une ligne à une colonne par slicing

- Soit le tableau suivant à 1 dimension : une ligne de 6 valeurs :

```
X = np.arange(1, 7)
X
array([1, 2, 3, 4, 5, 6])
```

- On peut le transformer en un tableau de 6 lignes à 1 valeurs à chaque fois.

⇒ Avec un reshape :

```
Y = X.reshape(6,1)
print(Y)
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

- Avec un slicing spécial :

```
Y = X[:, np.newaxis]
print(Y)
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

Exercices

Exercice

- Soit une matrice carrée de 10 de côté remplie régulièrement.
- On veut extraire la matrice avec uniquement les nombres impairs et les dizaines paires (1, 3, 5, 7, 9, 21, 23, etc.)
- Additionner le bloc de 3*3 partant de 11 avec celui partant de 35 + 1000
- Mettez les résultats dans la matrice carré de départ, dans le bloc de 3 partant de 35.

Exercice masque zeros reshape

Soit le vecteur [1, 2, 3, 4, 5].

On veut construire un nouveau vecteur avec 3 zéros entre chaque valeur.

On fera finalement un code avec un nombre de zéros variable (mais au moins 1)

Solution : <http://bliaudet.free.fr/IMG/txt/MasqueZerosReshape.py>

Précisions sur le reshaping : changement de dimension

Principes

- On peut réorganiser les données d'un tableau en changeant les lignes et les colonnes.
- On doit forcément avoir le même nombre d'éléments dans le tableau de départ et dans le tableau d'arrivée :
⇒ Par exemple, un tableau (4, 4) peut être réorganisé en (2, 8) ou (16).
- Le reshaping définit une vue.

➤ Code :

```
# creation d'un tableau de 4 lignes et 4 colonnes
a = np.random.randint(1, 10, size=(4,4))

# tranformation en un tableau de 2 lignes et 8 colonnes
b = a.reshape(2, 8)

a[2, 2]=99
b[0,4] # vaut 99
```

Méthode inverse : « aplatis » un tableau : ravel()

```
cube=np.ones((2,3,2))
cube.shape # (2, 3, 2)
a=cube.ravel() # équivalent à .flatten()
a.shape # (12) : c'est le nombre d'éléments
a
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

- C'est équivalent à un reshape à une dimension :

```
aa=cube.reshape(cube.size)
aa
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```


Variante : création et reshape enchainées :

➤ *Code précédent, enchainée :*

```
np.random.randint(1, 10, size=(4,4)).reshape(2,8)
```

➤ *Version plus subtile*

- Avec 2 boucles en compréhension, on récupère des indices de matrice qui nous permettent de calculer des valeurs :

⇒ i=0 : 0, 1, 2, 3, 4

⇒ i=1 : 10, 11, 12, 13, 14

⇒ etc.

```
np.array( [ (10*i+j) for i in range(5) for j in range(5) ] )  
array([ 0,  1,  2,  3,  4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30,  
       31, 32, 33, 34, 40, 41, 42, 43, 44])
```

- On reshape en matrice 5, 5 :

```
np.array( [ (10*i+j) for i in range(5) for j in range(5) ]  
) .reshape(5, 5)  
array([[ 0,  1,  2,  3,  4],  
       [10, 11, 12, 13, 14],  
       [20, 21, 22, 23, 24],  
       [30, 31, 32, 33, 34],  
       [40, 41, 42, 43, 44]])
```

Exercices

Exercice reshape de base

- Aplatissez le cube précédent avec un reshape.
- Aplatissez n'importe quel tableau avec un reshape

Exercice Matrice

- On reprend les exercices Matrice et Cube :
- On veut remplir une matrice de 3 lignes et 2 colonnes avec des valeurs de 1 à n en la remplissant comme avec un arange(), ligne par ligne.

⇒ Contenu du tableau :

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

⇒ Solution : <http://bliaudet.free.fr/IMG/txt/arangeReshapeMatrice.py>

Exercice Cube

- On reprend les exercices Matrice et Cube :
- On veut remplir un cube de côté 2 avec des valeurs de 1 à n en le remplissant progressivement niveau par niveau et ligne par ligne dans chaque niveau.
- Attention : quand on reshape à 3 dimensions, la troisième dimension est en 1^{er} indice ! On a un tableau (z, ligne, colonne)

⇒ Contenu du tableau :

```
array([[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]])
```

⇒ Solution : <http://bliaudet.free.fr/IMG/txt/arangeReshapeCube.py>

- On peut mettre à jour l'algorithme avec les boucles pour que la 3^{ème} dimension soit en 1^{er} indice :

⇒ Solution : <http://bliaudet.free.fr/IMG/txt/arangeCubeZFirst.py>

Les tableaux booléens

Principes

- On peut avoir des tableaux numpy avec uniquement des valeurs booléennes.
- True vaut aussi 1 et False vaut 0. Ainsi, on peut faire des sommes.

Création d'un tableau de booléens avec une opération de vectorisation

- On commence par se créer un tableau de températures de -5 à 20 pour le mois de mars :

```
mars_temp = np.random.randint(-5, 20, size=31)
```

- résultats :

```
array([19, 12, -2, 19, 8, 11, -3, 10, 6, -1, 7, -4, 2, 7, 9, 16,
       8, -2, 11, -5, 2, 0, 5, 14, 6, -4, 10, 2, 8, 18, -4])
```

- L'expression : `mars_temp > 0` retourne un tableau de booléen pour chaque valeur du tableau initial.

➤ *code*

```
mars_temp > 0
```

- On peut utiliser tous les opérateurs de comparaisons.
- Ce sont des opérations vectorisées.
- Elles produisent un tableau de booléens qui sont très pratiques.

➤ *résultats*

```
array([True , True , False, True , True , True , False, True ,
       True , False, True , False, True , True , True , True ,
       True , False, True , False, True , False, True , True ,
       True , False, True , True , True , True , False])
```

- On va regarder ensuite des usages de ces tableaux.

Sum, any et all dans un tableau de booléens

- *Nombre de jours à température >0 en mars :*

```
np.sum(mars_temp >0)
```

⇒ Le résultat vaut 22

⇒ L'expression est très compacte et assez intuitive syntaxiquement.

⇒ On peut utiliser tous les opérateurs de comparaisons.

- *Y a-t-il eu au moins un jour en mars à plus que 18 degrés ? La réponse est True.*

<https://numpy.org/doc/stable/reference/generated/numpy.any.html>

```
np.any(mars_temp >18)
```

- *Y a-t-il eu au moins un jour en mars à 20 degrés ? La réponse est False .*

<https://numpy.org/doc/stable/reference/generated/numpy.any.html>

```
np.any(mars_temp == 20)
```

- *Combien a-t-il eu de jour en mars à plus que 18 degrés ? La réponse est 2*

```
np.sum(mars_temp >18)
```

- *Est-ce que tous les jours du mois de mars ont été >0 degré ? La réponse est False.*

<https://numpy.org/doc/stable/reference/generated/numpy.all.html>

```
np.all(mars_temp >0)
```

Rappels : mean, max, min, sum, size

<https://numpy.org/doc/stable/reference/generated/numpy.mean.html#numpy.mean>

- On peut appliquer les fonctions sum, mean (moyenne), min et max, size (count) à un tableau :

```
np.sum(mars_temp)
```

& et | avec les tableaux de booléens combinés entre eux

➤ *Nombres de jours >10 degrés après le 15 mars ?*

- Il faut créer un tableau des jours de mars :

```
mars_jours = np.arange(1,mars_temp.size +1, dtype=np.int8)
```

- On va faire un & sur les température > 10 et sur les jours > 15, et on somme le tout :

```
np.sum( (mars_temp >10) & (mars_jours>15) )
```

➤ *Nombre de jours après le 17 mars au-dessus de 15 degrés ou au-dessous de 5 degrés :*

```
np.sum( ((mars_temp<5) | (mars_temp >15)) & (mars_jours>17) )
```

- ⇒ A noter que le & n'est pas un && : c'est un opérateur « bitwise » : bit à bit : c'est plus rapide.
- ⇒ A noter que le & est prioritaire sur tout : il faut donc parenthéser ses termes.
- ⇒ Le | est prioritaire sur les comparaisons : il faut parenthéser ses termes.

Compléments sur all() et any()

```
np.all(True)           # True
np.all(1)               # True
np.all('0')             # False
np.any(1)               # True
np.any('0')             # False
np.all('hello'=='hello') # True
np.any(5==5)            # True
```

```
np.all([1, 1, 0] )      # False
```

```
np.any([1, 1, 0] )      # True
```

```
np.all([True, False, True] ) # False
```

```
np.all(['1', '1', '0'] )  # False
```

```
np.any([1, 1, '0'] )      # True
```

```
np.zeros(5).any()         # False
```

```
np.ones(5).all()          # True
```

Présentation

- Les indexations avancées vont nous permettre d'**extraire certaines valeurs du tableau**, comme avec un slice.
- L'extraction se fait avec une **comparaison vectorisée**.
- **Le résultat est un nouveau tableau** : ce n'est pas une vue comme avec un slice.

Exemples

➤ *Les températures de mars > 10 degrés*

- On récupère un tableau de booléens : sont à true les températures >10
 - Ce tableau est en paramètre du tableau complet : ça ne garde que les valeurs qui sont à true
- ⇒ Code :

```
mars_temp[ mars_temp>10 ]
```

➤ *Les températures après le 17 mars, un jour sur 2*

⇒ Code :

```
mars_temp[ (mars_jours>17) & (mars_jours %2 == 0) ]
```

⇒ Code alternatif avec un slice :

```
mars_temp[17::2]
```

➤ *Les températures après le 17 mars 1 jour sur 2, supérieures à 10 degrés :*

```
mars_temp[  
    (mars_jours>17) &  
    (mars_jours %2 == 0) &  
    (mars_temps>10)  
]
```

Modification de certaines valeurs

- On veut remplacer les valeur négatives par la moyenne des valeurs positives.
- C'est une pratique courante de nettoyage des données : on supprime ou on remplace les valeurs aberrantes. Ici on considère que le valeurs négatives sont aberrantes.

➤ *Code*

- On doit d'abord définir la valeur de remplacement : ce sera la moyenne pour les valeurs positives :

```
moy = np.mean(mars_temp[mars_temp>=0])
```

- Ensuite on remplace les valeurs négatives par cette valeur

```
mars_temp[mars_temp<0] = moy
```

Création d'un tableau par indexation

- On peut créer un tableau à partir des éléments indexés d'un autre tableau :
- Création d'un tableau de départ :

```
tab = np.array(['spam', 'bean', 'eggs'], dtype=str)
```

- Création d'un nouveau tableau :

```
tab_spam=tab[ [0, 0, 0, 1, 1, 0, 2, 2] ]
```

⇒ A noter qu'il y a 2 crochets.

- Résultats du nouveau tableau :

```
['spam', 'spam', 'spam', 'bean', 'bean', 'spam', 'eggs', 'eggs']
```


np.where() : selection

```
# initialisation
import numpy as np
a=np.arange(10)
```

➤ *Sélection :*

Pour sélectionner certaines valeurs, on peut écrire :

```
res=np.where(a>6)
print(res)
print(type(res))
print(res[0])
```

Ca retourne la liste des valeurs concernées, dans un tuple.

```
(array([7, 8, 9]),)
<class 'tuple'>
[7 8 9]
```

Pour avoir directement un tableau, on écrit :

```
res=np.where(a>6)[0]
print(res)
print(res.dtype)
print(type(res))
print(res.size)
print(res[0])
```

```
[7 8 9]
int64
<class 'numpy.ndarray'>
3
7
```

➤ *Modification :*

On utilise le where surtout pour une modification :

```
res=np.where(a>6, 99, a)
print(res)
print(res.dtype)
print(res.shape)
```

```
[ 0  1  2  3  4  5  6 99 99 99]
int64
(10,)
```

➤ **Modification subtile :**

```
import numpy as np
a=np.arange(5).reshape(5, 1)
b=np.array([88, 99])
res=np.where(a>6, b, a)
print(b)
print(res)
print(res.dtype)
print(res.shape)
```

```
[88 99]
```

```
[[0 0]
```

```
 [1 1]
```

```
 [2 2]
```

```
 [3 3]
```

```
 [4 4]]
```

```
int64
```

```
(5, 2)
```

⇒ Ici, on a un broadcast entre a et b

Exercices .

Dans un tableau de 100 entiers entre 1 et 10, comptez combien de 10, de 9, de >6
Remplacez les 9 par 99

Vectorisation (11p - 76-86)

- On mettra nos exemples dans des notebooks.
- Tous les codes de ce chapitre sont dans un notebook dont l'image HTML est ici : <http://bliaudet.free.fr/notebook/NoteBook-Numpy-3-vectorisation.html>

Présentation

- La vectorisation est une technique de calcul particulière appliquée au ndarray.
- La vectorisation consiste à appliquer une opération au tableau et que cette opération s'applique en fait à tous ses éléments.
- La vectorisation permet une simplification d'écriture et une optimisation majeure du temps de traitement.
- Les opérations de vectorisation créent un nouvel objet.

Principes algorithmiques

➤ *Pour faire la même chose, on a 3 techniques :*

- ⇒ Boucles d'algorithmique classique
- ⇒ Compréhension de liste ou générateur
- ⇒ Vectorisation

➤ *En algorithmique classique, on écrit :*

```
Y = []  
for x in X:  
    Y.append(maf(x))
```

➤ *En compréhension de liste, on écrit :*

```
Y = [ maf(x) for x in X]
```

⇒ Avec un générateur, on écrit (le générateur économise la mémoire)

```
Y = ( maf(x) for x in X)
```

➤ *En numpy, on doit penser encore plus court, grâce à la vectorisation :*

```
Y = maf(X)
```

Exemple et constatation de l'optimisation

```
import numpy as np  
a = np.arange(1000)
```

- Calcul classique avec une écriture en compréhension :

```
%timeit [x**2 + 2*x - 1 for x in a]
```

⇒ 560 μ s \pm 33.1 μ s per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

- Calcul numpy avec vectorisation :

```
%timeit a**2 + 2*a - 1
```

⇒ 5.48 μ s \pm 232 ns per loop (mean \pm std. dev. of 7 runs, 100,000 loops each)

⇒ C'est 100 fois plus rapide !!!

- Tous les opérateurs de calcul de base sont vectorisés.
- Les fonctions de calcul de base sont aussi vectorisées : cf. exemple ci-dessous avec np.sqrt()

ufunc

- Le mécanisme général qui applique une fonction à un tableau est connu sous le terme de :
⇒ universal function ou ufunc.
- Ça peut être utile avec les moteurs de recherche.
⇒ <https://numpy.org/doc/stable/reference/ufuncs.html>

Multiplication de matrices « traditionnelle »

Attention ! Ca n'a rien à voir !

- Attention ! La multiplication de matrice est une opération mathématique qu'il ne faut pas confondre avec le calcul vectoriel.
- Le calcul vectoriel permet de faire une opération identique sur tous les points d'un tableau (ou de plusieurs s'ils ont le même format).
- La multiplication de deux matrices se définit ainsi :
 - ⇒ Avec
 - A matrice nA lignes, nk colonnes.
 - B matrice nk lignes, ncB colonnes.
 - ⇒ $C = A \times B$: X est le signe de multiplication de matrice
 - C matrice nA lignes, ncB colonnes telle que :

$$C_{i,j} = \text{somme}(k=0, nk) A_{i,k} * B_{k,j}$$

→ Cette formule est une formule d'algorithmique classique : on peut la coder en Python

Fonction dot() : multiplication de matrice

- En numpy, la multiplication de deux matrices se fait avec la méthode « dot() »

```
C=np.dot(A, B)
```

Utilisation d'un paramètre en sortie

```
import numpy as np
a = np.arange(1, 1_000_000, dtype=np.float64)
%timeit np.sqrt(a)
```

⇒ 2.13 ms ± 78.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

- Par défaut, les fonctions vectorisées écrivent dans un nouvel objet.
- On peut préciser le tableau en sortie avec le paramètre « out ». Ainsi, on écrit dans un tableau qui existe déjà. Ca économise de la place et le temps de création du tableau.

```
%timeit np.sqrt(a, out=a)
```

⇒ 1.24 ms ± 4.01 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

⇒ C'est 2 fois plus rapide : c'est toujours ça !

Remarques sur le timeit

➤ Utilisation d'options

- Pour accélérer les tests, on peut ajouter 2 options dans le timeit : -r 1 -n 1
 - ⇒ Ca permet de n'avoir qu'un tour de boucle. Ça donne un résultat moins précis, mais on garde les ordres de grandeur.
- Documentation timeit : <https://docs.python.org/fr/3/library/timeit.html>

```
import numpy as np
a = np.arange(1, 1_000_000, dtype=np.float64)
%timeit -r 1 -n 1 [np.sqrt(x) for x in a]
```

⇒ 1.35 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

```
%timeit -r 1 -n 1 np.sqrt(a)
```

⇒ 8.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

```
%timeit -r 1 -n 1 np.sqrt(a, out=a)
```

⇒ 1.9 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

➤ timeit dans l'interpréteur python

- On peut utiliser le timeit dans l'interpréteur python.
 - ⇒ On utilise la fonction timeit.timeit(). On lui passe tout le code en paramètre.
 - ⇒ C'est plus lourd.
- Documentation timeit : <https://docs.python.org/fr/3/library/timeit.html>

```
import timeit

timeit.timeit('L=list(range(1000));[x**2 for x in L]',
number=10000)

timeit.timeit('import numpy as
np;L=list(range(1000));a=np.array(L);[x**2 for x in a]',
number=10000)

timeit.timeit('import numpy as
np;L=list(range(1000));a=np.array(L);a**2', number=10000)
```

Modification de certains éléments : méthode at()

- Certaines fonctions vectorisées donnent accès à la méthode « at() ».
- Cette méthode permet de modifier les éléments d'un tableau qui seront identifiés par leur indice dans une liste.

⇒ Notez que normalement, la méthode sqrt ne modifie pas le tableau en entrée.

⇒ Avec le at(), les index fournis seront modifiés.

```
import numpy as np
a = np.arange(10, dtype=np.float32)
# array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.], dtype=float32)

# que les racines de a[1], a[4] et a[9]
np.sqrt.at(a, [1, 4, 9])
a
array([0., 1., 2., 3., 2., 5., 6., 7., 8., 3.], dtype=float32)
```

Statistiques par colonne ou par ligne : agrégation

Principes

- Les fonctions de statistique (sum, mean, min, max, size) s'appliquent à tout le tableau par défaut, quel que soit les dimensions du tableau.
- On va pouvoir appliquer ces fonctions pour toutes les colonnes ou pour toutes les lignes.
⇒ Par colonne, c'est l'équivalent de statistiques de base en SQL.

Esemple

- Un tableau de 3 lignes et 3 colonnes :

```
a = np.arange(1, 10).reshape(3, 3)
```

1	2	3
4	5	6
7	8	9

- Somme de tous les éléments du tableau :

```
np.sum(a) # vaut 45
```

- Somme de tous les éléments de chaque colonne :

```
np.sum(a, axis=0)
```

ou

```
a.sum(axis=0)
```

1	2	3
4	5	6
7	8	9

12	15	18
----	----	----

- Somme de tous les éléments de chaque ligne :

```
np.sum(a, axis=1)
```

ou

```
a.sum(axis=1)
```

1	2	3
4	5	6
7	8	9

6
15
24

Principes

- **np.nan** est une valeur de type float qui veut dire « not a number » et qui veut aussi dire qu'on n'a pas l'information.
- Le nan est l'équivalent du NULL en SQL.
- Comme en SQL, tout calcul avec un nan (un NULL en SQL) retourne un nan (un NULL en SQL).
⇒ nan est équivalent à : « inconnu ». Si on fait un calcul avec un « inconnu » le résultat est toujours « inconnu ».
- On peut contourner en numpy. Toutes les fonctions numpy ont des variantes qui ignorent le nan : on précède leur nom de « nan ». Par exemple : **nanmean()**.
- **np.isnan(var)** permet de savoir si la valeur de « var » vaut nan
- **np.invert(var)** permet d'inverser une valeur ou une série de valeurs booléennes.
⇒ C'est équivalent à **logical_not(var)** qui gère mieux la transformation des entiers

Exemple 1

```
a = np.arange(1, 10, dtype=np.float64).reshape(3, 3)
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
```

```
np.sum(a)      # 45
```

```
a[1, 1] = np.nan
a
array([[ 1.,  2.,  3.],
       [ 4., nan,  6.],
       [ 7.,  8.,  9.]])
```

```
np.sum(a)      # nan
np.Nansum(a)   # 40
```

```
np.sum(a, axis=0)
array([12., nan, 18.])
```

```
np.nansum(a, axis=0)
array([12., 10., 18.])
```

Compter les éléments nan ou les non nan d'un tableau

- Pour compter les nan, ou les non nan, on commence par transformer le tableau en tableau de booléens : True si nan, False sinon avec la fonction `isnan()`
- Ensuite, on somme les booléens : ça somme donc le nombre de nan.
- Pour compter les non nan, il suffit d'inverser le tableau de booléens avec la fonction `invert()`.

⇒ Nombre de nan :

```
a = np.arange(1, 10, dtype=np.float16)
a[0] = np.nan
np.sum(np.isnan(a)) # 1
```

⇒ Nombre de non nan :

```
np.sum(np.invert(np.isnan(a))) # 8
```

⇒ Cas avec une matrice :

```
a = np.arange(1, 10, dtype=np.float16).reshape(3,3)
a[0,0] = np.nan
np.sum(np.isnan(a), axis=0) # array([1, 0, 0])

np.sum(np.invert(np.isnan(a)), axis=0) # array([2, 3, 3])
```

Écrire nos propres fonctions vectorisées

Principes

- Les fonctions vectorisées sont beaucoup plus rapides et plus pratiques pour coder.
- Les fonctions vectorisées python sont écrites en C.
- On peut écrire nos propres fonctions vectorisées.
 - ⇒ On peut les écrire en C : <https://docs.python.org/fr/dev/extending/extending.html>
 - ⇒ On peut aussi écrire nos propres fonctions vectorisées en utilisant « cython » et « numba ».
 - Cython : https://cython.readthedocs.io/en/latest/src/userguide/numpy_tutorial.html
 - Numba : <https://numba.pydata.org/>

Décorateur @np.vectorize

- On a intérêt à ajouter le décorateur @np.vectorize devant nos fonctions scalaires qu'on veut utiliser en vectorisation.

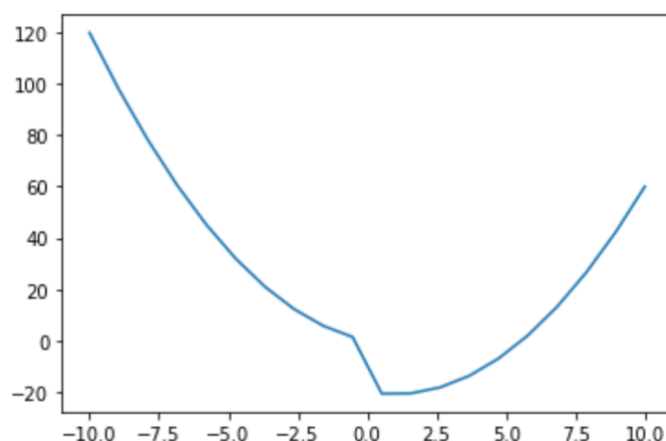
⇒ Exemple :

```
import numpy as np
import matplotlib.pyplot as plt

@np.vectorize
def maf(x):
    return x**2 - 2*x + (0 if x <= 0 else -20)
X=np.linspace(-10, +10, 20)
Y=maf(X) # c'est de la programmation vectorielle

# on met un point-virgule à la fin de la dernière ligne pour
éviter un bruit
plt.plot(X, Y);
```

→ Ici le @np.vectorize est nécessaire du fait du « if » dans le return.



On voit le « -20 » quand on passe au X positifs.

Le jeu de données fait que le passage est en pente : en réalité, il doit être vertical.

- On mettra nos exemples dans des notebooks.
- Tous les codes de ce chapitre sont dans un notebook dont l'image HTML est ici : <http://bliaudet.free.fr/notebook/NoteBook-Numpy-4-broadcasting.html>

Principes

- Le broadcasting concerne les opérations sur des tableaux de tailles différentes.
- Il faut être prudent avec le broadcasting car il peut être très couteux en temps et en mémoire.
- Le broadcasting est difficile à comprendre sur des tableaux à plus de 2 dimensions.

Exemples

- En python : `5 * [1, 2, 3]` : ça donne `[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]`
⇒ Ca duplique (*5) la série du tableau.
- En python : `[2, 4, 6] * [1, 2, 3]` : ça donne une erreur
- En numpy : `5 * np.array([1, 2, 3])` : ça donne `[5, 10, 15]`
⇒ La multiplication est faite sur chaque élément du tableau.
- En numpy : `np.array([2, 4, 6])*np.array([1, 2, 3])` : ça donne `[2, 8, 18]`
⇒ La multiplication se fait élément par élément de même indice
- En numpy : `np.array[5] * np.array([1, 2, 3])` : ça donne `[5, 10, 15]`
⇒ **Le premier tableau est « broadcasté »** : on fait comme s'il valait `[5, 5, 5]`
→ Du coup, on peut faire une multiplication élément par élément.
- En numpy : `np.array([2, 4])*np.array([1, 2, 3])` : ça donne une erreur !!
⇒ Le broadcasting n'est pas possible.
- En numpy : `np.array([[2], [4]])*np.array([1, 2, 3])` : ça donne `[[2, 4, 6], [4, 8, 12]]`

⇒ On va voir dans quelles conditions il sera possible.

⇒ Le bilan est que c'est possible si les dimensions de même nature sont identiques ou valent 1

Broadcasting entre matrices

Principe général

- Pour que le broadcasting soit possible entre 2 matrices, il faut que soit les dimensions de même axe soient identiques ou que l'une soit égale à 1.

Exemple 1 :

```
a=np.array([10, 20, 30 ])
b=np.array([ [1, 1, 1], [2, 2, 2], [3, 3, 3] ])
```

a*b c'est :

10, 20, 30 * 1, 1, 1
 2, 2, 2
 3, 3, 3

La ligne est propagée :

10, 20, 30 * 1, 1, 1
10, 20, 30 2, 2, 2
10, 20, 30 3, 3, 3

La multiplication peut alors se faire élément par élément.

Le résultat est donc :

10, 20, 30 * 1, 1, 1 = 10, 20, 30
10, 20, 30 2, 2, 2. 20, 40, 60
10, 20, 30 3, 3, 3 30, 60, 90

Exemple 2 :

```
a=np.array([10, 20, 30 ]).reshape(3,1)
b=np.array([ [1, 1, 1], [2, 2, 2], [3, 3, 3] ])
```

a*b c'est :

```
10  * 1, 1, 1
20   2, 2, 2
30   3, 3, 3
```

La colonne est propagée :

```
10, 10, 10 * 1, 1, 1
20, 20, 20   2, 2, 2
30, 30, 30   3, 3, 3
```

La multiplication peut alors se faire élément par élément.

Le résultat est donc :

```
10, 10, 10 * 1, 1, 1 = 10, 10, 10
20, 20, 20   2, 2, 2   40, 40, 40
30, 30, 30   3, 3, 3   90, 90, 90
```

Exemple 3 :

```
a=np.array([1, 2, 3 ]).reshape(3,1)
b=np.array([4, 5 ]).reshape(1,2)
```

a*b c'est :

```
1 * 4,5
2
3
```

La colonne de « a » est propagée :

```
1,1 * 4,5
2,2
3,3
```

Les lignes de « b » sont propagées :

```
1,1 * 4,5
2,2   4,5
3,3   4,5
```

La multiplication peut alors se faire élément par élément.

Le résultat est donc :

```
1,1 * 4,5 = 4, 5
2,2   4,5 = 8, 10
3,3   4,5 = 12, 15
```

Application de broadcasting

Valeurs possibles pour 2 dés

- Valeurs pour 1 dés :

```
X = np.arange(1, 7)
X
array([1, 2, 3, 4, 5, 6])
```

```
X[0] # 1
X[1] # 2
```

- Valeurs pour le 2^{ème} dés : on met la ligne en colonne

```
Y = X[:, np.newaxis]
Y
```

```
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
Y[0,0] # 1
Y[0,1] # 2
```

```
des_2 = X + Y
des_2
```

```
array([[ 2,  3,  4,  5,  6,  7],
       [ 3,  4,  5,  6,  7,  8],
       [ 4,  5,  6,  7,  8,  9],
       [ 5,  6,  7,  8,  9, 10],
       [ 6,  7,  8,  9, 10, 11],
       [ 7,  8,  9, 10, 11, 12]])
```

Fonction unique() : nombre d'occurrences de chaque valeur dans un tableau

```
a=des_2
res=np.unique(a, return_counts=True)
res
(array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]),
 array([1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1]))
```

```
values=res[0]
freq=res[1]
array([1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1])
array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

Valeurs possibles pour 3 dés

- Valeurs pour le 3^{ème} dé : on fait à Y ce qu'on a fait à X :

```
Z = Y[:, np.newaxis]
```

```
Z
```

```
array([[1]],  
       [[2]],  
       [[3]],  
       [[4]],  
       [[5]],  
       [[6]])
```

```
Z[0,0,0] # 1
```

```
Z[0,0,1] # 2
```

```
des_3 = X + Y + Z
```

```
des_3
```

```
array([[ 3,  4,  5,  6,  7,  8],  
       [ 4,  5,  6,  7,  8,  9],  
       [ 5,  6,  7,  8,  9, 10],  
       [ 6,  7,  8,  9, 10, 11],  
       [ 7,  8,  9, 10, 11, 12],  
       [ 8,  9, 10, 11, 12, 13]],  
  
       [[ 4,  5,  6,  7,  8,  9],  
       [ 5,  6,  7,  8,  9, 10],  
       [ 6,  7,  8,  9, 10, 11],  
       [ 7,  8,  9, 10, 11, 12],  
       [ 8,  9, 10, 11, 12, 13],  
       [ 9, 10, 11, 12, 13, 14]],  
  
       [[ 5,  6,  7,  8,  9, 10],  
       [ 6,  7,  8,  9, 10, 11],  
       [ 7,  8,  9, 10, 11, 12],  
       [ 8,  9, 10, 11, 12, 13],  
       [ 9, 10, 11, 12, 13, 14],  
       [10, 11, 12, 13, 14, 15]],  
  
       [[ 6,  7,  8,  9, 10, 11],  
       [ 7,  8,  9, 10, 11, 12],  
       [ 8,  9, 10, 11, 12, 13],  
       [ 9, 10, 11, 12, 13, 14],  
       [10, 11, 12, 13, 14, 15],  
       [11, 12, 13, 14, 15, 16]],  
  
       [[ 7,  8,  9, 10, 11, 12],  
       [ 8,  9, 10, 11, 12, 13],  
       [ 9, 10, 11, 12, 13, 14],  
       [10, 11, 12, 13, 14, 15],  
       [11, 12, 13, 14, 15, 16],  
       [12, 13, 14, 15, 16, 17]],  
  
       [[ 8,  9, 10, 11, 12, 13],  
       [ 9, 10, 11, 12, 13, 14],  
       [10, 11, 12, 13, 14, 15],  
       [11, 12, 13, 14, 15, 16],  
       [12, 13, 14, 15, 16, 17],  
       [13, 14, 15, 16, 17, 18]])
```

Exercices

Exercice

Quelle est la valeur de la fréquence la plus élevée ?

Solution : <http://bliaudet.free.fr/IMG/txt/valeurDeLaFrequenceLaPlusElevee.py>

On fera aussi une version pour 3 dés (cf ci-dessous).

Exercice : Le damier .

- Ecrire une fonction damier qui crée une matrice carrée remplie comme un damier avec des entiers 0 et 1.
- L'indice de la première valeur de la matrice (coordonnées 0-0) pour être 0 ou 1 en fonction d'un paramètre passé à la fonction. Par défaut, ce sera 1.
- On fera une version Python et une version Numpy
- Pour la version Python, il suffit d'avoir 2 boucles imbriquées et de remplir une liste en faisant attention à créer la matrice (liste de listes) et à la valeur 1 ou 0 selon les cas. C'est assez facile à faire.
- Pour la version numpy, on peut commencer par remplir une matrice de 1 à n, puis la transformer en booléen selon que les entiers sont pairs ou impairs. Ensuite, il faut inverser seulement les lignes impaires, si nécessaire.

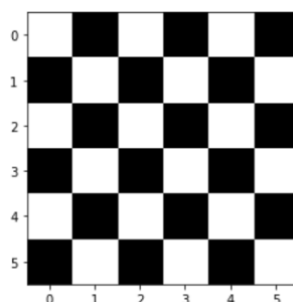
- Pour l'affichage du damier, si damier est notre matrice de 0 et de 1, on écrit :

```
import numpy as np
import matplotlib.pyplot as plt

damier=np.array([[1,0],[0,1]])
plt.imshow(damier.astype(float), cmap='gray');
```

- On écrira une fonction d'affichage avec la taille du damier en paramètre et la valeur de la case (0,0) si nécessaire.
- On peut aussi faire une version numpy en utilisant la fonction indices() qui retourne 2 tableaux. En sommant les 2, on obtiendra facilement le résultat.

⇒ Solution : <http://bliaudet.free.fr/IMG/txt/damierPyhtonEtNumpy.py>



Exercice : la fonction `fij` : `tab[i, j] = 100*i + 10*j + offset` .

- On vous demande d'écrire une fonction « `fij` » qui crée un tableau de `nl` lignes et `nc` colonnes, composé d'entiers, tels que `tab[i, j] = 100*i + 10*j + offset`.
- On fera une version Python et une version Numpy.
- La version Python est simple.
- La version numpy utilise le broadcasting et la vectorisation :

```
# tab[i, j] = 100*i + 10*j

#                               4 colonnes
#           [[0],      *   [0, 1, 2, 3, 4]
#           [1],
# 3 lignes  [2]]
```

⇒ Solution : <http://bliaudet.free.fr/IMG/txt/fijPythonEtNumpy.py>

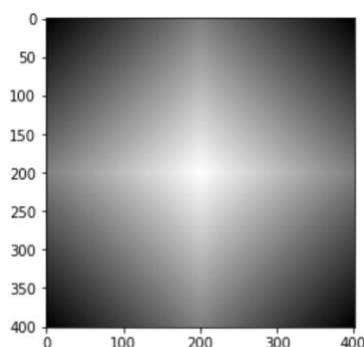
Exercice : la pyramide .

- construire un tableau en pyramyde :

```
0 1 2 3 2 1 0
1 2 3 4 3 2 1
2 3 4 5 4 3 2
3 4 5 6 5 4 3
2 3 4 5 4 3 2
1 2 3 4 3 2 1
0 1 2 3 2 1 0
```

- C'est une matrice carrée. Le principe que sur chaque ligne, comme sur chaque colonne, on monte de 1 (ou un paramètre au choix), jusqu'au milieu, puis on redescend. On a forcément un nombre impair d'éléments par ligne et par colonne.
- On vous demande d'écrire une fonction pyramyde qui crée ce tableau.
- Le plus pratique sera de passer en paramètre un demi-coté qu'on appelle « taille ».
- Aux quatre coins du tableau on trouve la valeur 0 . Dans la case centrale on trouve la valeur $2*taille$.
- On fera une version semi-python : avec un reshape numpy et des boucles python pour remplir.
- Et une version « pure » numpy. Méthodes : arange + concatenate + slicing d'inversion ligne/colonne + broadcasting
- On testera la vitesse d'exécution.
- On affichera l'image la pyramide.

⇒ Solution : <http://bliaudet.free.fr/IMG/txt/pyramPythonEtNumpy.py>



Fonction indices()

- La fonction `indices((nl, nc))` est « bizarre » :
⇒ elle permet de créer 2 tableaux de format (nl, nc)
- Chaque ligne du 1^{er} est remplie de la même valeur. Les lignes vont de 0 à nl non compris.
- Chaque colonne du 2^{ème} est remplie de la même valeur. Les colonnes vont de 0 à nc non compris.

```
ix, iy = np.indices ( (3, 4) )
```

```
ix
```

```
array([[0, 0, 0, 0],  
       [1, 1, 1, 1],  
       [2, 2, 2, 2]])
```

```
iy
```

```
array([[0, 1, 2, 3],  
       [0, 1, 2, 3],  
       [0, 1, 2, 3]])
```


Fonction meshgrid()

Principe

- La fonction meshgrid(a, b) s'applique à deux tableaux de dimension 1.
- Elle produit 2 tableaux qui sont les tableaux intermédiaires du broadcast de a et b mis à la verticale

Explication par un exemple :

```
a=np.arange(4)
a
```

```
array([0, 1, 2, 3])
```

```
b=np.arange(6,8)
b
```

```
array([6, 7])
```

```
A, B = np.meshgrid(a, b)
```

```
A
```

```
array([[0, 1, 2, 3],
       [0, 1, 2, 3]])
```

```
B
```

```
array([[6, 6, 6, 6],
       [7, 7, 7, 7]])
```

```
A*B
```

```
array([[ 0,  6, 12, 18],
       [ 0,  7, 14, 21]])
```

- C'est pareil que :

```
A = a*np.ones(b.size, dtype=int).reshape(b.size, 1)
```

```
B = b.reshape(b.size, 1)*np.ones(a.size, dtype=int)
```

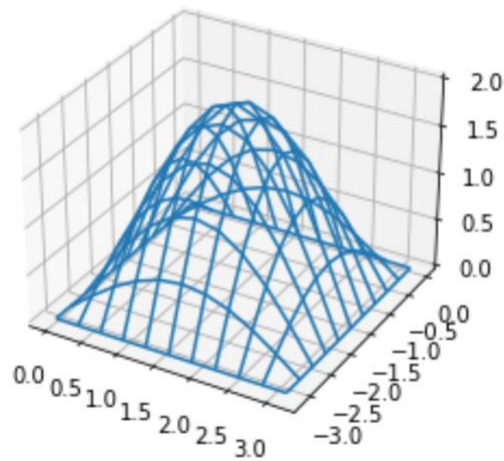
Exemple plus subtil :

```
import numpy as np
import matplotlib.pyplot as plt

Xticks, Yticks = (
    np.linspace(0, np.pi, num=11),
    np.linspace(-np.pi, 0, num=11)
)

X, Y = np.meshgrid(Xticks, Yticks)
Z = np.cos(X+Y)-np.cos(X-Y)

import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_wireframe(X, Y, Z);
```



Fonction linspace()

Variante 1

<https://numpy.org/doc/stable/reference/generated/numpy.linspace.html?highlight=linspace#numpy.linspace>

- La fonction `linspace(début, fin, nb_éléments)` permet de créer une liste de `nb_éléments` entre `début` et `fin` comprise et répartis à intervalles réguliers. Les éléments sont des réels.

```
a=np.linspace(1, 9, 5)
a # array([1., 3., 5., 7., 9.])

b=np.linspace(0, 5, 5)
b # array([0. , 1.25, 2.5 , 3.75, 5.  ])
```

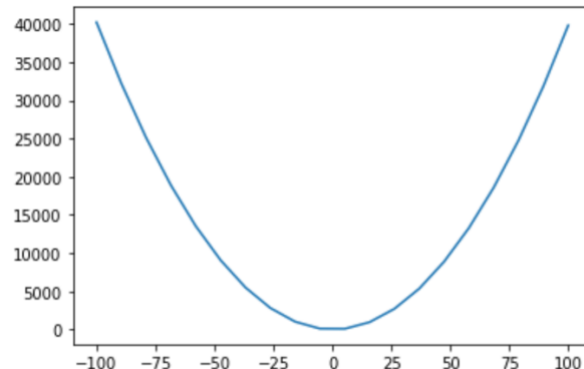
- C'est pratique pour dessiner une fonction entre 2 bornes.

⇒ Par exemple : la fonction $4x^2-2x+3$ entre -100 et +100.

```
import numpy as np
import matplotlib.pyplot as plt

def maf(x):
    return 4*x**2 -2*x + (1 if x <=0 else 10)
X=np.linspace(-100, +100, 20)
Y=maf(X) # c'est de la programmation vectorielle

# on met un point-virgule à la fin de la dernière ligne pour
éviter un bruit
plt.plot(X, Y);
```

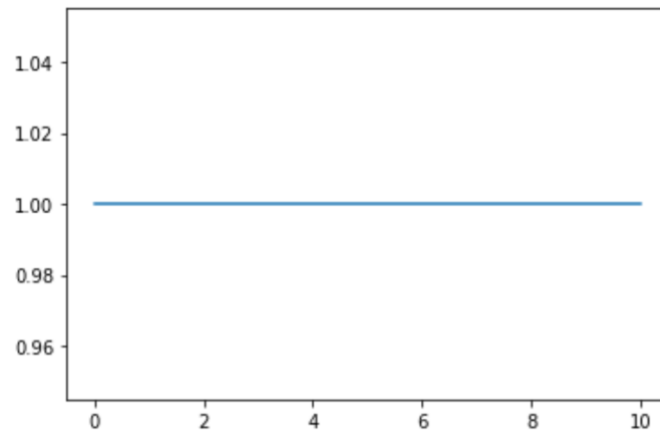


Variante 2

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(0., 10., 50)
Y = np.cos(X)**2 + np.sin(X)**2 # vaut toujours 1 par définition

plt.plot(X, Y);
```



Variante 3 : avec le décorateur @np.vectorize

- On a intérêt à ajouter le décorateur @np.vectorize devant nos fonctions scalaires qu'on veut utiliser en vectorisation.

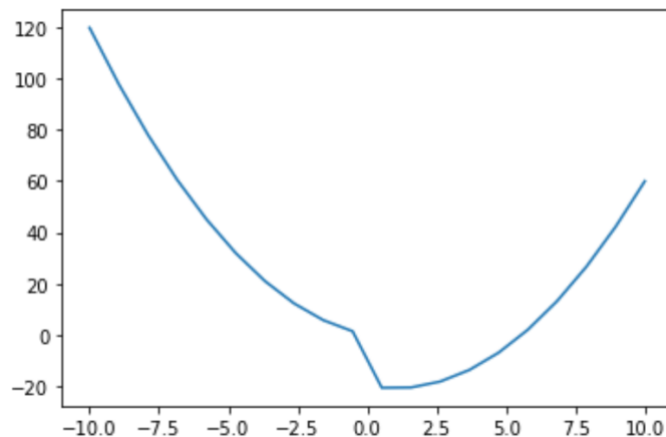
⇒ Exemple :

```
import numpy as np
import matplotlib.pyplot as plt

@np.vectorize
def maf(x):
    return x**2 - 2*x + (0 if x <=0 else -20)
X=np.linspace(-10, +10, 20)
Y=maf(X) # c'est de la programmation vectorielle

# on met un point-virgule à la fin de la dernière ligne pour
éviter un bruit
plt.plot(X, Y);
```

→ Ici le @np.vectorize est nécessaire du fait du « if » dans le return.



On voit le « -20 » quand on passe au X positifs.

Le jeu de données fait que le passage est en pente : en réalité, il doit être vertical.

Notion de masque

On va afficher une image d'un centre noir qui va en dégradé vers le gris.

- Pour ça, on va contruire une matrice carré avec un 0 au centre et un dégradé

```
width = 128
center = width // 2

ix, iy = np.indices((width, width))
image = (ix-center)**2 + (iy-center)**2
```

```
image[center, center] # 0
```

- Si on faisait :

```
image = (ix-center) + (iy-center)
```

⇒ on aurait :

```
array([[ -256,  -255,  -254, ...,   -3,   -2,   -1],
       [ -255,  -254,  -253, ...,   -2,   -1,    0],
       [ -254,  -253,  -252, ...,   -1,    0,    1],
       ...,
       [   -3,   -2,   -1, ...,  250,  251,  252],
       [   -2,   -1,    0, ...,  251,  252,  253],
       [   -1,    0,    1, ...,  252,  253,  254]])
```

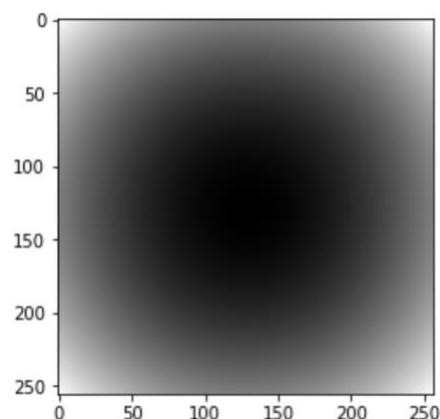
⇒ On le met au carré, ça conne un résultat plus régulier

- Pour afficher l'image, on utilise la fonction `imshow()` :

```
import matplotlib.pyplot as plt
plt.imshow(image, cmap='gray');
```

- A noter que dans les notebooks, on écrit plutôt :

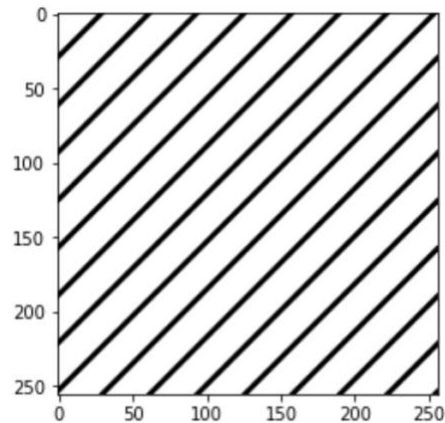
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
plt.imshow(image, cmap='gray');
```



On va afficher des rayures noires en diagonale

- Pour ça, on reprend les tableaux ix et iy (des 0, 1, 2, etc. jusqu'à 255), en lignes et en colonnes.
- On les ajoute : 0, 1, 2, etc, puis 1, 2, 3, etc, ... jusqu'à 255, 256, 257, etc.
- On fait %32 : le reste sera de 0 à 31
- Et on test si c'est <=26
- On aura 6 False tout les 32, en diagonale.

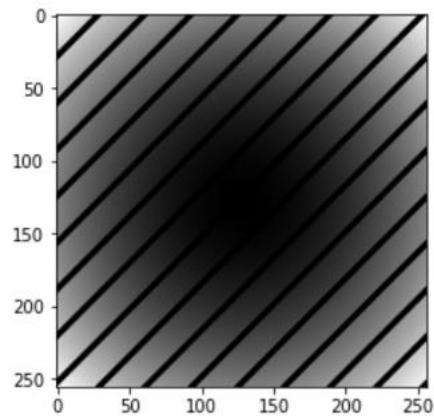
```
rayures = (ix + iy) % 32 <= 26  
plt.imshow(rayures, cmap='gray');
```



Les diagonales vont servir de masque sur l’affichage de départ

- Il suffit de multiplier les deux matrices : rayures * images

```
plt.imshow(rayures*image, cmap='gray');
```



Listes de fonctions plus ou moins subtiles :

```
ix, iy = np.indices((2, 3))
ix, iy = np.indices((2, 3))      # ix : [ [0, 0, 0],
                                   #       [1, 1, 1] ]
                                   # iy : [ [0, 1, 2],
                                   #       [0, 1, 2] ]

a=np.arange(4)                    # a : [0, 1, 2, 3]
b=np.arange(6,8)                  # b : [6, 7]

# A et B ont le même format
# b est mis en colonne
A, B = np.meshgrid(a, b)          # A : [ [0, 1, 2, 3],
                                   #       [0, 1, 2, 3] ]
                                   # B : [ [6, 6, 6, 6],
                                   #       [7, 7, 7, 7] ]

# ajouter une bordure autour d'un tableau
Z = np.ones((5,5))
Z = np.pad(Z, pad_width=2, mode='constant', constant_values=99)

# multiplication de matrices
A=np.arange(8).reshape(2,4)
B=np.arange(12).reshape(4,3)
C=np.dot(A, B)                    # shape : (2, 3)
```

3 – MATPLOTLIB

Références

Matplotlib

Il y a beaucoup de références sur internet.

On peut utiliser directement le site officiel (en le traduisant en français si nécessaire).

- Matplotlib : <https://matplotlib.org/>
- pour la dimension 2 : https://matplotlib.org/2.0.2/users/pyplot_tutorial.html ;
- pour la dimension 3 : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html.

MOOC

<https://lms.fun-mooc.fr/courses/course-v1:UCA+107001+session02/courseware/82ac208ce93b479f90a5a65e04753f5a/29dff594499b4875b945ce02e0ec8740/>

Bases

- Ressources disponibles en ligne en anglais (à traduire automatiquement) :
⇒ pour la dimension 2 : https://matplotlib.org/2.0.2/users/pyplot_tutorial.html
- On va utiliser des extraits tels quels.

Imports habituels

```
import numpy as np
import matplotlib.pyplot as plt
```

- On ne va pas utiliser plt.ion() : c'est mode automatique de **matplotlib** dans les notebooks.
- On veut apprendre à utiliser **matplotlib** dans un contexte normal.

Changer la taille par défaut des figures matplotlib

```
plt.rcParams["figure.figsize"] = (6, 6)
```

⇒ https://matplotlib.org/stable/users/prev_whats_new/dflt_style_changes.html

plt.plot

➤ Bases

- Si on donne seulement *une* liste de valeurs, elles sont considérées comme les Y
⇒ les X étant les entiers en nombre suffisant et en commençant à 0.

In []:

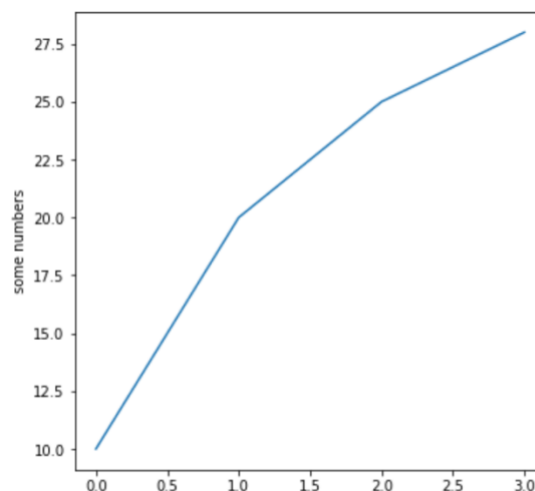
```
# si je ne donne qu'une seule liste à plot
# alors ce sont les Y
import matplotlib.pyplot as plt
plt.plot([10, 20, 25, 28]) ;
```

⇒ Le « ; » évite un bruit.

```
# on peut aussi facilement ajouter une légende
# ici sur l'axe des y
import matplotlib.pyplot as plt
plt.ylabel('some numbers')
plt.plot([10, 20, 25, 28]) ;
```

```
# pour afficher, à la fin, c'est mieux :
plt.show()
```

➤ Résultats



➤ Paramétrages : 'b', 'ro', etc

- On peut changer le style utilisé par plot pour tracer :

```
import matplotlib.pyplot as plt
plt.plot([10, 20, 25, 28], r-') ;
```

⇒ par défaut 'b-' : ligne bleue (b pour bleu, et - pour ligne).

⇒ 'ro' : rouge rond (point)

⇒ 'g--' : vert pointillé

⇒ 'bs' : bleu carré (square)

⇒ 'g^' : vert triangle

→ Voyez [la documentation de référence de plot](#) pour une liste complète.

➤ **Exemple de base**

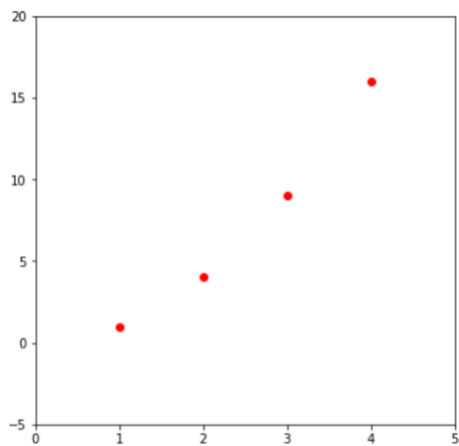
```
import numpy as np
import matplotlib.pyplot as plt

# on fixe la taille de l'affichage
plt.rcParams["figure.figsize"] = (6, 6)

# Le plus souvent on passe à plot
# une liste de X ET une liste de Y
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 16, 25], 'ro')

# on va utiliser l'axe des X : entre 0 et 5
# l'axe des Y : entre -5 et 20
plt.axis([0, 5, -5, 20])

plt.show()
```



➤ **Mise en page : dessiner plusieurs fonctions :**

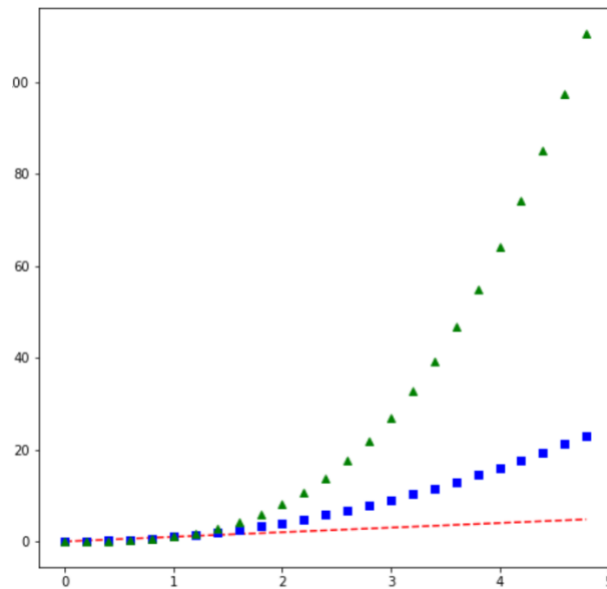
```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (8, 8)

# échantillon de points entre 0 et 5 espacés de 0.2
t = np.arange(0., 5., 0.2)

# plusieurs styles de ligne
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
# on pourrait ajouter d'autres plot bien sûr aussi

plt.show()
```



Plusieurs subplots : `plt.subplot`

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (15, 6)

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

## deux domaines presque identiques
# celui-ci pour les points bleus
t1 = np.arange(0.0, 5.0, 0.1)
# celui-ci pour la ligne bleue
t2 = np.arange(0.0, 5.0, 0.02)

# on crée un 'subplot' et on dessine dedans
plt.subplot(211) # plt.subplot(2, 1, 1) : 2 lignes, 2 colonnes,
# 1er emplacement.
plt.axis([0, 5, -1, 1.5])
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'r')

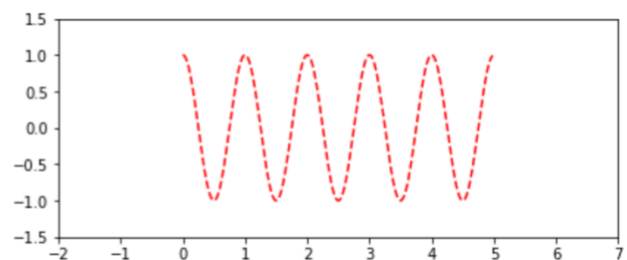
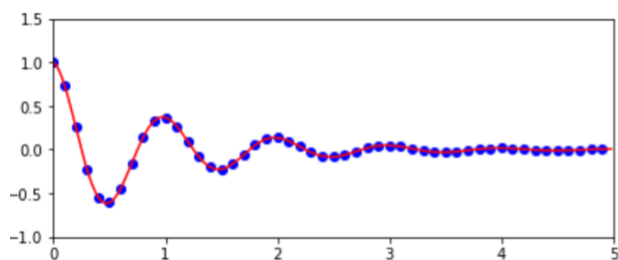
# une deuxième subplot et on dessine dedans
plt.subplot(222) # plt.subplot(2, 1, 1) : 2 lignes, 2 colonnes,
# 2eme emplacement.
plt.axis([-2, 7, -1.5, 1.5])
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')

plt.show()

plt.show()
```

⇒ Avec les paramètres de subplot (211), on peut travailler sur 2 colonnes et dessiner les blocs sur une même ligne.

⇒ Associé à `plt.rcParams["figure.figsize"]`, on peut tout faire !



Mise en page : option 1 => à éviter

- On crée des figures numérotées : `plt.figure(1)`
⇒ Dans les figures on met les subplots
- On peut revenir sur une figure et sur un subplot pour y ajouter par exemple un titre.
⇒ Ca change l'ordre !!! (si on ne le fait pas, ce n'est pas le même ordre).
⇒ Ce n'est pas très pratique.

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (6, 6)

plt.figure(1)                # the first figure

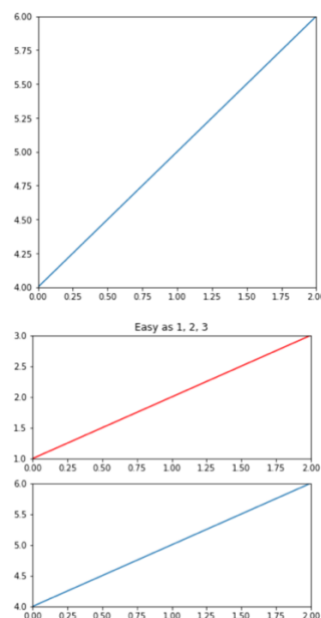
plt.subplot(211)             # the first subplot in the first figure
plt.axis([0, 2, 1, 3])
plt.plot([1, 2, 3], 'r')

plt.subplot(212)             # the second subplot in the first figure
plt.axis([0, 2, 4, 6])
plt.plot([4, 5, 6])

plt.figure(2)                # a second figure
plt.axis([0, 2, 4, 6])
plt.plot([4, 5, 6])          # creates a subplot(111) by default

plt.figure(1)                # figure 1 current; subplot(212) current
plt.subplot(211)             # make subplot(211) in figure1 current
plt.title('Easy as 1, 2, 3') # subplot 211 title

plt.show()
```



Mise en page : option 2 => à utiliser

- Traditionnellement les subplots sont appelés 'axes'.
- On va créer en une instruction : la figure et ses subplots (ici sur 2 lignes et 1 colonne) :

```
fig1, (ax1, ax2) = plt.subplots(2, 1)
```

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (6, 6)

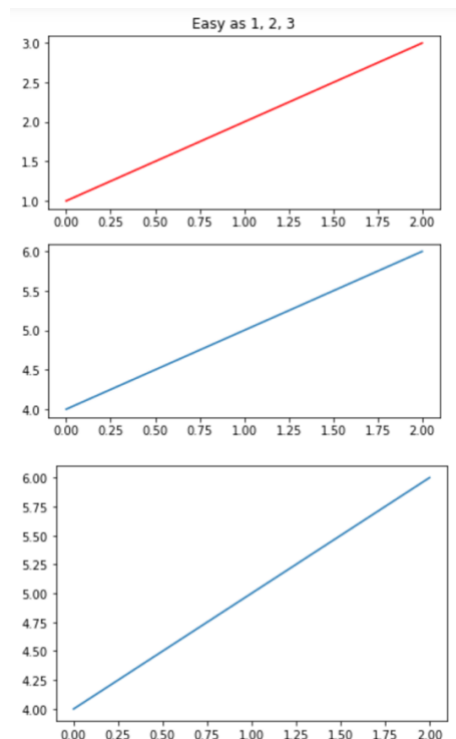
# On crée en une instruction : la figure et ses subplots
fig1, (ax1, ax2) = plt.subplots(2, 1)

# la méthode plot s'applique directement à un subplot
ax1.plot([1, 2, 3], 'r')
ax2.plot([4, 5, 6])

fig2, ax3 = plt.subplots(1, 1)
ax3.plot([4, 5, 6])
fig2.set_size_inches(6,4) # pour réduire la taille

# pour revenir au premier subplot : on utilise ax1
# attention la méthode plt.title() devient ax1.set_title()
ax1.set_title('Easy as 1, 2, 3')

plt.show()
```



Variantes d'affichage

plt.hist

- La fonction `plt.hist()` permet d'afficher un histogramme.

⇒ https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (6, 6)

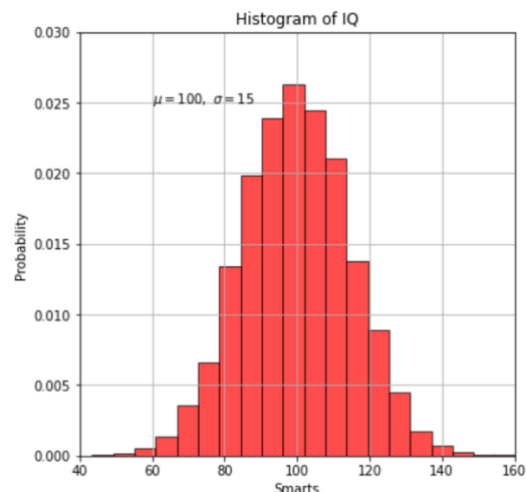
# on bricole une formule pour avoir des valeurs distribuées selon
une loi normale (courbe de Gauss)
np.random.seed(19680801) # on fixe la graine du générateur
aléatoire
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# on dessine un histogramme avec 20 barres (bins)
# plt.hist(x, 20 , density=1) # version minimale
plt.hist(x, 20 , density=1, color='r', edgecolor = 'black',
alpha=0.7)

# on rajoute des décorations
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')

plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)

plt.show()
```



plt.scatter

- La fonction `plt.scatter()` permet d'afficher des points colorés.

⇒ https://matplotlib.org/stable/api/as_gen/matplotlib.pyplot.scatter.html

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (5, 5)

# on fixe la graine du générateur aléatoire
np.random.seed(19680801)

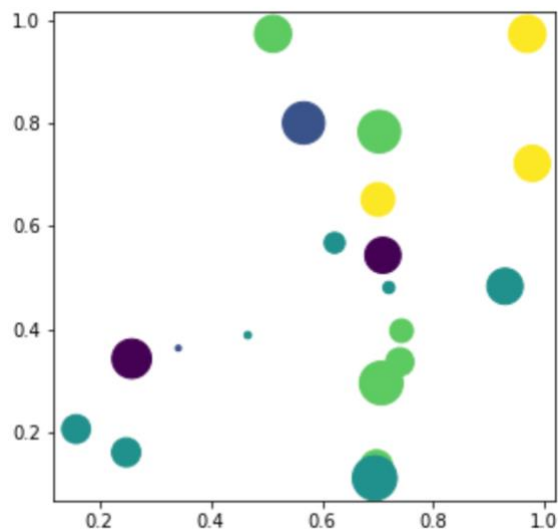
N = 20                                     # nombre de point
x = np.random.rand(N)                     # abscisses des points : x
y = np.random.rand(N)                     # ordonnées des points : y

colors = np.random.rand(N)                # une couleur par point
colors = (colors*100).astype(int)%5/100    # pour réduire à
                                           # 5 couleurs (bricolage)

area = np.full(N, 50)                     # une taille de point par point, fixe
area = 3*(15*np.random.rand(N))**2       # une taille par point
                                           # aléatoire (bricolage)

plt.scatter(x, y, s=area, c=colors, alpha=1) # alpha:transparence

plt.show()
```



plt.boxplot

- Avec boxplot vous obtenez des boîtes « à moustache

⇒ https://matplotlib.org/stable/api/as_gen/matplotlib.pyplot.boxplot.html

```
import numpy as np
import matplotlib.pyplot as plt

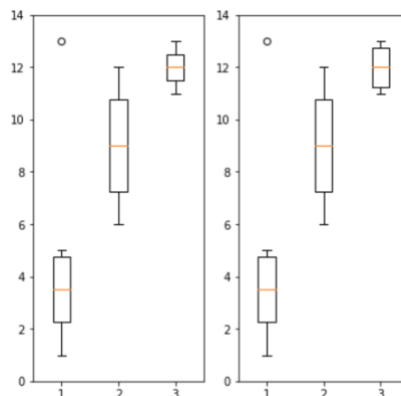
# disposition en 2 subplots en colonnes
fig, (ax1, ax2) = plt.subplots(1, 2)

# on peut passer à boxplot une liste de suites de nombres
# chaque suite donne lieu à une boîte à moustache
# ici 3 suites
ax1.boxplot([[1, 2, 3, 4, 5, 13], [6, 7, 8, 10, 11, 12], [11, 12, 13]]) # à noter le 13 isolé
ax1.set_ylim(0, 14)

# on peut passer un ndarray numpy
# attention : les colonnes forment les séries : 3 colonnes, 3
# séries. Il faut gérer les tailles différentes
a = np.array([[1, 6, 11],
              [2, 7, 12],
              [3, 8, 13],
              [4, 10, 11],
              [5, 11, 12],
              [13, 12, 13]])

ax2.boxplot(a)
ax2.set_ylim(0, 14)

plt.show()
```



<file:///Users/bertrandliaudet/Downloads/w7-s10-c2-matplotlib-3d.html>

Bases

- Ressources disponibles en ligne en anglais (à traduire automatiquement) :
⇒ pour la dimension 3 : https://matplotlib.org/2.0.2/mpl_toolkits/mplot3d/tutorial.html
- On va utiliser des extraits tels quels.

In []:

```
# la ratiion habituelle d'imports
import matplotlib.pyplot as plt
# et aussi numpy, même si ça n'est pas strictement nécessaire
import numpy as np
Pour pouvoir faire des visualisations en 3D, il vous faut importer ceci :
```

In []:

```
# même si l'on n'utilise pas explicitement
# d'attributs du module Axes3D
# cet import est nécessaire pour faire
# des visualisations en 3D
from mpl_toolkits.mplot3d import Axes3D
```

Dans ce notebook nous allons utiliser un mode de visualisation un peu plus élaboré, mieux intégré à l'environnement des notebooks :

In []:

```
# ce mode d'interaction va nous permettre de nous déplacer
# dans l'espace pour voir les courbes en 3D
# depuis plusieurs points de vue
%matplotlib notebook
```

Comme on va le voir très vite, avec ces réglages vous aurez la possibilité d'explorer interactivement les visualisations en 3D.

Un premier exemple : une courbe

Commençons par le premier exemple du tutorial, qui nous montre comment dessiner une ligne suivant une courbe définie de manière paramétrique (ici, xx et yy sont fonctions de zz). Les points importants sont :

- la composition d'un plot (plusieurs figures, chacune composée de plusieurs *subplots*), reste bien entendu valide ; j'ai enrichi l'exemple initial pour mélanger un *subplot* en 3D avec un *subplot* en 2D ;
- l'utilisation du paramètre `projection='3d'` lorsqu'on crée un *subplot* qui va se prêter à une visualisation en 3D ;
- l'objet *subplot* ainsi créé est une instance de la classe `Axes3DSubplot` ;
- on peut envoyer à cet objet :
 - la méthode `plot` qu'on avait déjà vue pour la dimension 2 (c'est ce que l'on fait dans ce premier exemple) ;
 - des méthodes spécifiques à la 3D, que l'on voit dans les exemples suivants.

In []:

```
# je choisis une taille raisonnable compte tenu de l'espace
# disponible dans fun-mooc
fig = plt.figure(figsize=(6, 3))

# voici la façon de créer un *subplot*
# qui se prête à une visualisation en 3D
```

```

ax = fig.add_subplot(121, projection='3d')

# à présent, copié de
# https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#line-plots
# on crée une courbe paramétrique
# où x et y sont fonctions de z
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
# on fait maintenant un appel à plot normal
# mais avec un troisième paramètre
ax.plot(x, y, z, label='parametric curve')
ax.legend()

# on peut tout à fait ajouter un plot usuel
# dans un subplot, comme on l'a vu pour la 2D
ax2 = fig.add_subplot(122)
x = np.linspace(0, 10)
y = x**2
ax2.plot(x, y)
plt.show()

```

Un autre point à remarquer est qu'avec le mode d'interaction que nous avons choisi :

%matplotlib notebook

vous bénéficiez d'un mode d'interaction plus riche avec la figure. Par exemple, vous pouvez cliquer dans la figure en 3D, et vous déplacer pour changer de point de vue ; par exemple si vous sélectionnez l'outil Pan/Zoom (l'outil avec 4 flèches), vous pouvez arriver à voir ceci :

Les différents boutons d'outil [sont décrits plus en détail ici](https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#scatter-plots). Je dois avouer ne pas arriver à tout utiliser lorsque la visualisation est faite dans un notebook, mais la possibilité de modifier le point de vue peut s'avérer intéressante pour explorer les données.

En explorant les autres exemples du tutorial, vous pouvez commencer à découvrir l'éventail des possibilités offertes par matplotlib.

Axes3DSubplot.scatter

Comme en dimension 2, scatter permet de montrer un nuage de points.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#scatter-plots

scatter3d_demo.py

In []:

```

'''
=====
3D scatterplot
=====

Demonstration of a basic scatterplot in 3D.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

```

```

import numpy as np

fig = plt.figure(figsize=(4, 4))

def randrange(n, vmin, vmax):
    '''
    Helper function to make an array of random numbers having shape (n, )
    with each number distributed Uniform(vmin, vmax).
    '''
    return (vmax - vmin)*np.random.rand(n) + vmin

ax = fig.add_subplot(111, projection='3d')

n = 100

# For each set of style and range settings, plot n random points in the box
# defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].
for c, m, zlow, zhigh in [('r', 'o', -50, -25), ('b', '^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zlow, zhigh)
    ax.scatter(xs, ys, zs, c=c, marker=m)

ax.set_xlabel('X Label')
plt.show()

```

In []:

Axes3DSubplot.plot_wireframe

Utilisez cette méthode pour dessiner en mode "fil de fer".

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#wireframe-plots.

wire3d_demo.py

```

from mpl_toolkits.mplot3d import axes3d

fig = plt.figure(figsize=(4, 4))

ax = fig.add_subplot(111, projection='3d')

# Grab some test data.
X, Y, Z = axes3d.get_test_data(0.05)

# Plot a basic wireframe.
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
plt.show()

```

In []:

In []:

Axes3DSubplot.plot_surface

Comme on s'en doute, plot_surface sert à dessiner des surfaces dans l'espace ; ces exemples montrent surtout comment utiliser des couleurs ou des *patterns*.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#surface-plots.

surface3d_demo.py

```

'''
=====
3D surface (color map)
=====
'''

```

In []:

Demonstrates plotting a 3D surface colored with the coolwarm color map.
The surface is made opaque by using antialiased=False.

Also demonstrates using the LinearLocator and custom formatting for the
z axis tick labels.

```
'''
```

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np
```

In []:

```
fig = plt.figure(figsize=(4, 4))

# dans le tuto on trouve
# fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))

# personnellement je trouve plus facile à retenir ceci
ax = fig.add_subplot(111, projection='3d')

# Make data.
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                      linewidth=0, antialiased=False)

# Customize the z axis.
ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
surface3d_demo2.py
```

In []:

```
'''
```

```
=====
3D surface (solid color)
=====
```

Demonstrates a very basic plot of a 3D surface using a solid color.

```
'''
```

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
```

In []:

```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
```



```
# Make data
u = np.linspace(0, 2 * np.pi, 30)
v = np.linspace(0, np.pi, 30)
x = 10 * np.outer(np.cos(u), np.sin(v))
y = 10 * np.outer(np.sin(u), np.sin(v))
z = 10 * np.outer(np.ones(np.size(u)), np.cos(v))
```

```
# Plot the surface
ax.plot_surface(x, y, z, color='b')
```

```
plt.show()
surface3d_demo3.py
```

In []:

```
'''
=====
3D surface (checkerboard)
=====
```

```
Demonstrates plotting a 3D surface colored in a checkerboard pattern.
'''
```

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator
import numpy as np
```

In []:

```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
```

```
# Make data.
X = np.arange(-5, 5, 0.25)
xlen = len(X)
Y = np.arange(-5, 5, 0.25)
ylen = len(Y)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
```

```
# Create an empty array of strings with the same shape as the meshgrid, and
# populate it with two colors in a checkerboard pattern.
```

```
colortuple = ('y', 'b')
colors = np.empty(X.shape, dtype=str)
for y in range(ylen):
    for x in range(xlen):
        colors[x, y] = colortuple[(x + y) % len(colortuple)]
```

```
# Plot the surface with face colors taken from the array we made.
surf = ax.plot_surface(X, Y, Z, facecolors=colors, linewidth=0)
```

```
# Customize the z axis.
ax.set_zlim(-1, 1)
ax.w_zaxis.set_major_locator(LinearLocator(6))
```

```
plt.show()
```

Axes3DSubplot.plot_trisurf

plot_trisurf se prête aussi au rendu de surfaces, mais sur la base de maillages en triangles.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#tri-surface-plots.

trisurf3d_demo.py

In []:

```
'''
=====
Triangular 3D surfaces
=====

Plot a 3D surface with a triangular mesh.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')

n_radii = 8
n_angles = 36

# Make radii and angles spaces (radius r=0 omitted to eliminate duplication).
radii = np.linspace(0.125, 1.0, n_radii)
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)

# Repeat all angles for each radius.
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)

# Convert polar (radii, angles) coords to cartesian (x, y) coords.
# (0, 0) is manually added at this stage, so there will be no duplicate
# points in the (x, y) plane.
x = np.append(0, (radii*np.cos(angles)).flatten())
y = np.append(0, (radii*np.sin(angles)).flatten())

# Compute z to make the pringle surface.
z = np.sin(-x*y)

ax.plot_trisurf(x, y, z, linewidth=0.2, antialiased=True)

plt.show()
trisurf3d_demo2.py
```

In []:

```
'''
=====
More triangular 3D surfaces
=====

Two additional examples of plotting surfaces with triangular mesh.

The first demonstrates use of plot_trisurf's triangles argument, and the
second sets a Triangulation object's mask and passes the object directly
to plot_trisurf.
'''

import numpy as np
```

In []:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as mtri

fig = plt.figure(figsize=(6, 3))

#=====
# First plot
#=====

# Make a mesh in the space of parameterisation variables u and v
u = np.linspace(0, 2.0 * np.pi, endpoint=True, num=50)
v = np.linspace(-0.5, 0.5, endpoint=True, num=10)
u, v = np.meshgrid(u, v)
u, v = u.flatten(), v.flatten()

# This is the Mobius mapping, taking a u, v pair and returning an x, y, z
# triple
x = (1 + 0.5 * v * np.cos(u / 2.0)) * np.cos(u)
y = (1 + 0.5 * v * np.cos(u / 2.0)) * np.sin(u)
z = 0.5 * v * np.sin(u / 2.0)

# Triangulate parameter space to determine the triangles
tri = mtri.Triangulation(u, v)

# Plot the surface. The triangles in parameter space determine which x, y, z
# points are connected by an edge.
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_trisurf(x, y, z, triangles=tri.triangles, cmap=plt.cm.Spectral)
ax.set_zlim(-1, 1)

#=====
# Second plot
#=====

# Make parameter spaces radii and angles.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi/n_angles

# Map radius, angle pairs to x, y, z points.
x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = mtri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid**2 + ymid**2 < min_radius**2, 1, 0)
```

```

triang.set_mask(mask)

# Plot the surface.
ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.plot_trisurf(triang, z, cmap=plt.cm.CMRmap)

plt.show()

```

Axes3DSubplot.contour

Pour dessiner des contours.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#contour-plots.

contour3d_demo.py

```

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

```

In []:

```

fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

```

In []:

```

plt.show()
contour3d_demo2.py

```

```

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

```

In []:

```

fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, extend3d=True, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

```

In []:

```

plt.show()
contour3d_demo3.py

```

```

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm

```

In []:

```

fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contour(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)

ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')

```

In []:

```
ax.set_zlim(-100, 100)
```

```
plt.show()
```

Axes3DSubplot.contourf

Comme Axes3DSubplot.contour, mais avec un rendu plein plutôt que sous forme de lignes (le f provient de l'anglais *filled*).

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#filled-contour-plots.

contourf3d_demo.py

In []:

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

In []:

```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contourf(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)
```

```
plt.show()
contourf3d_demo2.py
```

In []:

```
"""
.. versionadded:: 1.1.0
   This demo depends on new features added to contourf3d.
"""
```

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

In []:

```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contourf(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)
```

```
ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')
ax.set_zlim(-100, 100)
```

```
plt.show()
```

Axes3DSubplot.add_collection3d

Pour afficher des polygones.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#polygon-plots.

In []:

```
"""
=====
```

Generate polygons to fill under 3D line graph

=====

Demonstrate how to create polygons which fill the space under a line graph. In this example polygons are semi-transparent, creating a sort of 'jagged stained glass' effect.

"""

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
import matplotlib.pyplot as plt
from matplotlib import colors as mcolors
import numpy as np
```

In []:

```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
```

```
def cc(arg):
    return mcolors.to_rgba(arg, alpha=0.6)
```

```
xs = np.arange(0, 10, 0.4)
verts = []
zs = [0.0, 1.0, 2.0, 3.0]
for z in zs:
    ys = np.random.rand(len(xs))
    ys[0], ys[-1] = 0, 0
    verts.append(list(zip(xs, ys)))
```

```
poly = PolyCollection(verts, facecolors=[cc('r'), cc('g'), cc('b'),
                                         cc('y')])
```

```
poly.set_alpha(0.7)
ax.add_collection3d(poly, zs=zs, zdir='y')
```

```
ax.set_xlabel('X')
ax.set_xlim3d(0, 10)
ax.set_ylabel('Y')
ax.set_ylim3d(-1, 4)
ax.set_zlabel('Z')
ax.set_zlim3d(0, 1)
```

```
plt.show()
```

Axes3DSubplot.bar

Pour construire des diagrammes à barres.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#bar-plots.

bars3d_demo.py

In []:

"""

=====

Create 2D bar graphs in different planes

=====

Demonstrates making a 3D plot which has 2D bar graphs projected onto planes $y=0$, $y=1$, etc.

"""

```

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

```

In []:

```

fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
for c, z in zip(['r', 'g', 'b', 'y'], [30, 20, 10, 0]):
    xs = np.arange(20)
    ys = np.random.rand(20)

    # You can provide either a single color or an array. To demonstrate this,
    # the first bar of each set will be colored cyan.
    cs = [c] * len(xs)
    cs[0] = 'c'
    ax.bar(xs, ys, zs=z, zdir='y', color=cs, alpha=0.8)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

plt.show()

```

Axes3DSubplot.quiver

Pour afficher des champs de vecteurs sous forme de traits.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#quiver.

quiver3d_demo.py

In []:

```

'''
=====
3D quiver plot
=====

Demonstrates plotting directional arrows at points on a 3d meshgrid.
'''

```

```

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

```

In []:

```

fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')

# Make the grid
x, y, z = np.meshgrid(np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.8))

# Make the direction data for the arrows
u = np.sin(np.pi * x) * np.cos(np.pi * y) * np.cos(np.pi * z)
v = -np.cos(np.pi * x) * np.sin(np.pi * y) * np.cos(np.pi * z)
w = (np.sqrt(2.0 / 3.0) * np.cos(np.pi * x) * np.cos(np.pi * y) *
     np.sin(np.pi * z))

ax.quiver(x, y, z, u, v, w, length=0.1, normalize=True)

plt.show()

```

